The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

Vikhorev, Konstantin (2011) Real-time guarantees in high-level agent programming languages. PhD thesis, University of Nottingham.

Access from the University of Nottingham repository:
http://eprints.nottingham.ac.uk/13036/1/546491.pdf

Copyright and reuse:

The Nottingham ePrints service makes this work by researchers of the University of Nottingham available open access under the following conditions.

- Copyright and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners.
- To the extent reasonable and practicable the material made available in Nottingham ePrints has been checked for eligibility before being made available.
- Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.
- Quotations or similar reproductions must be sufficiently acknowledged.

Please see our full end user licence at:
http://eprints.nottingham.ac.uk/end_user_agreement.pdf

A note on versions:

The version presented here may differ from the published version or from the version of record. If you wish to cite this item you are advised to consult the publisher's version. Please see the repository url above for details on accessing the published version and note that access may require a subscription.

For more information, please contact eprints@nottingham.ac.uk

10 0659420 5

Real-time guarantees in high-level agent programming languages

by

Konstantin Vikhorev, MSc

Faculty of Science

School of Computer Science

THE UNIVERSITY OF NOTTINGHAM

Nottingham

Thesis submitted to the University of Nottingham

for the degree of Doctor of Philosophy

The University of Nottingham

GEORGE GREEN LIBRARY OF
SCIENCE AND ENGINEERING

JULY 2011

Abstract

In the thesis we present a new approach to providing soft real-time guarantees for Belief-Desire-Intention (BDI) agents. We analyse real-time guarantees for BDI agents and show how these can be achieved within a generic BDI programming framework.

As an illustration of our approach, we develop a new agent architecture, called AgentSpeak(RT), and its associated programming language, which allows the development of real-time BDI agents. AgentSpeak(RT) extends AgentSpeak(L) [28] intentions with *deadlines* which specify the time by which the agent should respond to an event, and *priorities* which specify the relative importance of responding to a particular event. The AgentSpeak(RT) interpreter commits to a priority-maximal set of intentions: a set of intentions that is maximally feasible while preferring higher priority intentions. Real-time tasks can be freely mixed with tasks for which no deadline and/or priority has been specified, and if no deadlines and priorities are specified, the behavior of the agent defaults to that of a non real-time BDI agent. We perform a detailed case study of the use of AgentSpeak(RT) to demonstrate its advantages. This case study involves the development of an intelligent control system for a simple model of a nuclear power plant. We also prove some properties of the AgentSpeak(RT) architecture such as guaranteed reactivity delay of the AgentSpeak(RT) interpreter and probabilistic guarantees of successful execution of intentions by their deadlines.

We extend the AgentSpeak(RT) architecture to allow the parallel execution of intentions. We present a multitasking approach to the parallel execution of intentions in the AgentSpeak(RT) architecture. We demonstrate advantages of parallel execution of intentions in AgentSpeak(RT) by showing how it improves behaviour of the intelligent control system for the nuclear power plant. We prove real-time guarantees of the extended AgentSpeak(RT) architecture.

We present a characterisation of real-time task environments for an agent, and describe how it relates to AgentSpeak(RT) execution time profiles for a plan and an action. We also show a relationship between the estimated execution time of a plan in a particular environment and the syntactic complexity of an agent program.

Acknowledgments

I would like to take this opportunity to acknowledge the guidance, advice, imparting of knowledge, feedback and immense level of support provided to me by my PhD supervisors:

- Dr. Brian Logan from the School of Computer Science at the University of Nottingham, and
- Dr. Natasha Alechina from the School of Computer Science at the University of Nottingham.

I deeply appreciate the financial support provided by the University of Nottingham and The Engineering and Physical Sciences Research Council (EPSRC) during my studies. Also I would like to acknowledge and single out a number of people for their advice, feedback and support over the course of my PhD:

- Dr. Peer-Olaf Siebers from the Intelligent Modelling & Analysis Research Group (IMA) at the University of Nottingham for his advice, suggestions, interest in my research, and his confidence in me.
- Mrs Christine Fletcher from the School of Computer Science at the University of Nottingham for her contributions as a Postgraduate Research Administrator.

I would like to acknowledge my post-graduate colleagues that are members of the Agents Lab at the University of Nottingham. In particular, I would like to thank my colleague Julian Zappala who provided me helpful comments and advices on the thesis.

Finally, but definitely not least I would like to acknowledge the support of my family: father Sergey Vikhorev, mother Galina Vikhoreva, younger brother Alexey Vikhorev, and two sisters Irina and Natasha who stuck by me and provided me with encouragement to keep on going.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

> *"There is a popular cliché ... which says that you cannot get out of computers any more than you have put in ..., that computers can only do exactly what you tell them to, and that therefore computers are never creative. This cliché is true only in a crashingly trivial sense, the same sense in which Shakespeare never wrote anything except what his first school teacher taught him to write — words."*
>
> Richard Dawkins, "The Blind Watchmaker", 1997

The design of computer systems that are capable of performing high-level management of user tasks, in complex dynamic environments, is very important in real-life commercial applications. Such computer systems includes high-level control of unmanned vehicles (e.g., Unmanned Arial Vehicles (UAV), Unmanned Surface Vehicles (USV), etc.), power plant control, telecommunications, financial and business processes, aircraft control, etc. New abstract mechanisms are required in order to specify, design, verify and implement these kinds of computer systems. One abstraction mechanism that is increasingly becoming accepted is the notion of *intelligent agents*:

software systems that act independently of direct external control for an undefined amount of time.

1.1 Intelligent Agents

Research into agent systems has yielded an extensive body of work in terms of both theoretical results and practical models and systems, and is still ongoing. While the term 'intelligent agent' or simply 'agent' is widely used by people working in closely related areas such as Engineering, Computer Science, Biology, and many others, there is no clear and universal definition of the 'agent'.

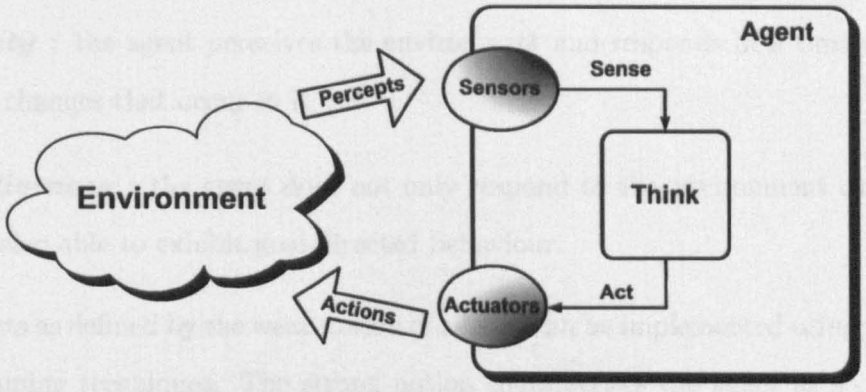

Figure 1.1: Agent and environment

For the purposes of this thesis, we define an *agent* as a hardware or software-based computer system situated in some environment. The agent is able to observe its environment via sensors. It reasons about changes in the environment, and performs actions via actuators in order to modify its environment. The view on the agent is shown in Figure 1.1. The agent environment can be physical (the physical world in case of robots) or software (graphical user interface, the Internet, etc.). While the agent can perform actions that change the environment, it has only partial control

over it.

Wooldridge and Jennings [43] distinguish weak and strong notions of agency, which describes properties of the agent. Their weak notion of agency defines an agent as a computer system that has following properties:

autonomy : the agent operates without direct human control over its actions and internal state;

social ability : the agent interacts with other agents and possibly with human operators using some communication language or interface e.g., they cooperate and coordinate their activities in order to accomplish their goals;

reactivity : the agent perceives the environment and responds in a timely fashion to changes that occur in it;

pro-activeness : the agent does not only respond to the environment changes, it is also able to exhibit goal-directed behaviour.

Agents as defined by the weak notion of agency can be implemented using standard programming techniques. The strong notion characterises the agent as a computer system which has the properties identified above and is either conceptualised or implemented using concepts that are usually applied to humans. It is very common in AI to characterise an agent using mentalistic notions such as knowledge, belief, desire, intention, and obligations. Also use of AI techniques such as learning or planning is mandatory attribute of the strong notion of the agent.

An agent is implemented and executed within a software framework called the *agent architecture*. An agent program is often implemented using a high level agent programming language associated with the agent architecture. The agent structure in that case consists of the agent architecture and the agent program:

```
agent = agent architecture + agent program.
```

There are several software models for programming agents. One commonly used agent software model of human-like agent reasoning is the 'Belief-Desire-Intention'(BDI) model [8]. The behaviour of a BDI agent is described using the notions of beliefs, desires and intentions.

1.2 The Problem of Real-Time Agency

The design of agent systems which can operate effectively in a dynamic environment is a major challenge for multiagent research. The reasoning processes implicit in many agent architectures can require significant time to execute, with the result that the environment may change while the agent makes a decision about which activity to pursue. Thus a decision made by the agent may be 'wrong' (e.g., incorrect, sub-optimal, or simply irrelevant) if it is not made in a timely manner. An agent in such an environment is *real-time* in the sense that correctness of the system depends not only on the response produced but the time at which it is produced.

In a real-time environment, the events to which the agent must respond should be characterized by a *deadline*, e.g., the time by which a goal must be achieved or the agent must respond to a change in its beliefs about the environment. In such an environment, a rational agent should not adopt an intention which it believes cannot be successfully executed by its deadline or continue to execute an intention after its deadline. For example, an agent should not adopt an intention of writing a research proposal which must be submitted by 4pm on Friday if there is insufficient time to write the proposal. Also, if the agent is unable to respond to all events by their deadlines, it should adopt intentions for the highest priority events which are feasible.

A number of agent architectures and platforms have been proposed for the development of agent systems which must operate in highly dynamic environments. For example, the Procedural Reasoning System (PRS) [15] and PRS-like systems, e.g., PRS-CL [26], JAM [18], SPARK [25] have features such as metalevel reasoning which facilitate the development of agents for real time environments. However, to provide real time guarantees, these systems have to be programmed for each particular task environment—there are no general methods or tools which allow the agent developer to specify that a particular goal should be achieved by a specified time or that an action should be performed within a particular interval of an event occurring.

Other architectures such as the Soft Real-Time Agent Architecture (SRTA) [37] and AgentSpeak(XL) [3] implement high-level agent frameworks with built-in real-time capabilities. However, although they allow the specification of time constraints and complex interactions between tasks, these agent architectures do not offer guarantees regarding the successful execution of tasks.

We now can introduce a statement of the thesis.

Thesis Statement. *High-level declarative agent programming languages based on the BDI paradigm have been the focus of considerable research in the multi-agent system community. However, while such languages provide powerful abstractions to facilitate the programming of complex patterns of agent interaction, they fail to capture a key element of agency, namely the ability to respond to changes in the agent's environment in a timely manner. The thesis will explore ways of adding soft real-time guarantees to high-level declarative agent programming languages such as AgentSpeak [28] and PRS [15]. This will involve developing ways to more precisely specify the execution cycle of such languages and enhancing BDI model specifications.*

1.3 Aims and Objectives

The aim of the thesis is to develop a new approach to providing soft real-time guarantees for Belief-Desire-Intention (BDI) agents that can be applied to a wide range of BDI-based agent programming languages.

The objectives of the thesis are:

1. to define what is meant by real-time guarantees for BDI agents and propose modifications to a generic BDI architecture to support real-time BDI agents;
2. to design and implement a BDI architecture which allows the development of real-time agents;
3. to prove real-time properties of that architecture, including guaranteed reactivity delay of the architecture and probabilistic guarantees of successful execution of intentions by their deadlines;
4. to characterise real-time task environments for an agent and to analyse their influence on the behaviour of an agent.

1.4 Thesis Structure

In this section we provide an overview of the thesis structure.

Chapter 2. Real-Time BDI agents. This chapter reviews the BDI agent model. Also the chapter presents the analysis of real-time guarantees for BDI agents and how these can be achieved within a BDI programming framework.

Chapter 3. Literature review. This chapter discusses different agent architectures, which can operate in a dynamic environment. The chapter distinguish

two types of agent architectures: the one with embedded real-time control and the one with cooperative real-time control. Furthermore the chapter evaluates the ability of each agent architecture to satisfy required real-time constraints.

Chapter 4. Changes to the BDI Architecture. This chapter explains the changes that must be made to a generic BDI architecture to implement a real-time BDI agent. The chapter distinguishes two types of changes: additional information about goals and plans to support real-time guarantees, and the extension of the BDI execution cycle to ensure that the agent's cycle time is bounded and the agent accomplishes a priority-maximal set of intentions with a specified level of confidence.

Chapter 5. AgentSpeak(RT): A Real-Time Agent Programming Language. This chapter presents AgentSpeak(RT), a programming language for real-time BDI agents. The chapter presents the syntax of AgentSpeak(RT) and describes the execution cycle of the AgentSpeak(RT) architecture. It also presents the example AgentSpeak(RT) agent program for control of a nuclear power plant.

Chapter 6. AgentSpeak(RT) Properties. This chapter gives the proof of guaranteed reaction time of the AgentSpeak(RT) interpreter and probabilistic guarantees of successful execution of intentions, i.e., an AgentSpeak(RT) agent is a real-time BDI agent. Also it presents a simple model of the 'difficulty' of the agent's task environment, and describes how to determine a probability that an intention of given priority will not be displaced from the schedule and will be executed by its deadline.

Chapter 7. AgentSpeak(RT) with Parallel Execution of Intentions. This chapter presents a multitasking approach to parallel execution of intentions in

the AgentSpeak(RT) architecture. The chapter discusses proofs of the real-time properties of the extended AgentSpeak(RT) architecture and presents an extended model of the environment 'difficulty' which allows parallel execution of intentions. The probability that in the extended model of a dynamic environment a scheduled intention of given priority will be successfully completed by its deadline is derived. Finally we demonstrate advantages of parallel execution of intentions in AgentSpeak(RT) by showing how it improves the behaviour of the example intelligent agent from Chapter 5.

Chapter 8. AgentSpeak(RT) Environment Characterisation. This chapter presents a characterisation of real-time task environments for an agent, and describes their implementation in AgentSpeak(RT). The chapter also describes a relationship between the estimated execution time of a plan in a particular environment and the syntactic complexity of agent programs.

Chapter 9. Conclusions and Future Work. This chapter provides a discussion and summary of the research including an evaluation of potential application domains. The major contributions and limitations of the work are described as well as outlining the areas and directions in which this research could be extended.

Chapter 2

Real-Time BDI agents

In this chapter we review the Belief-Desire-Intention (BDI) model for programming intelligent agents. We present analysis of real-time guarantees for BDI agents and demonstrate how these can be achieved within a BDI programming framework.

2.1 BDI Model

One commonly used agent software model is the Belief-Desire-Intention model in which an agent's behaviour is described using the notions of beliefs, desires and intentions. The origins of the model lie in the philosophy theory of human-like reasoning introduced by Michael Bratman [8]. Most well-known agent architectures like the Procedural Reasoning System (PRS) [15], AgentSpeak(L) [28], Jason [5], JAM [18], 3APL [17], 2APL [11], etc. are based on the BDI model.

The components of the BDI model can be briefly described as follows:

- *Beliefs* represent the informational model of the agent's world. Beliefs can also include inference rules and ground facts, which are updated by an agent's sensors. Rules are usually not updated by the sensors. The term *belief* is used,

rather than knowledge, because an agent's beliefs may not necessarily be true and can be changed in the future. Usually beliefs are represented as standard first-order predicates.

- *Desires* (or goals) represent states which the agent wishes to bring about. Goals may be consistent or inconsistent. Goals are inconsistent if they are mutually exclusive (i.e., achieving one implies not achieving another). Otherwise, the goals are consistent.

  There are two aspects of goals [42]: *declarative* (i.e., goal-to-be) which describes a desired state; and *procedural* (i.e., goal-to-do) describes a set of plans for achieving the goal.

- *Intentions* represent the agent's choice of particular courses of action i.e., plans, that is, desires which the agent has chosen to pursue in the current situation. Plans specify sequences of actions and subgoals an agent can use to achieve its goals given its beliefs.

The BDI model also incorporates another component – an input events queue. An *event* is a trigger for reactive activity by the agent. An event may update a belief base state, trigger plans or establish a new goal. There are two types of events: external, which are generated from outside and received by sensors, and internal, which are generated by the agent program.

A generic BDI architecture has been developed without much attention to reasoning time constraints. However the reasoning process implicit in BDI agent architectures can require significant time to execute, with the result that the environment may change while the agent makes a decision about which activity to pursue. Thus a decision made by the agent may be wrong (incorrect, sub-optimal, or simply irrelevant) if it is not made in a timely manner. This fact represents a gap that current

Algorithm 2.1 The BDI Interpreter [29]

1. initialize state of the agent
2. **repeat**
   - 2.1 update events using its sensors;
   - 2.2 deliberate over new events and choose a course of action (or plans) to fulfill them;
   - 2.3 update the intention structure by these plans
   - 2.4 execute an intention
3. **end repeat**

research is trying to address. Below, we provide a definition of real-time guarantees for a BDI agent.

2.2 Real-time Guarantees

Real-world intelligent systems perceive their environments and affect them through execution of actions. The environment imposes real-time constraints on such systems, which operate on this environment, i.e., the system has to have a bounded response time. For example, a post delivery robot moving in building corridors needs to respond quickly enough to avoid collisions with obstacles.

In real-time programming a distinction is made between *hard real-time* and *soft real-time* systems. In the context of agent systems, hard real-time means that the agent must process its inputs (i.e., facts and goals) and produce a response within a specified time. For an agent system, which provides hard real-time guarantees, there

is therefore a strict upper bound on the time to process incoming information and produce a response.

In soft real-time, the agent may not produce a response within the specified time in all cases, i.e., timeliness constraints may be violated under load and fault conditions without critical consequences[1]. For BDI agents, a relevant notion of 'response' is the adoption and successful execution of an intention i.e., the achievement of a high-level goal or responding to a change in the agent's environment.

We assume that each of the agent's intentions is associated with a (possibly infinite) *deadline* which specifies the time by which the goal should be achieved or a change in the agent's environment responded to, and an *expected execution time* (which depends on the agent's plan to achieve the goal or respond to the event and the state of the agent's environment). A set of intentions which can all be achieved by their deadlines is termed *feasible*. Which sets of intentions are feasible will depend on the speed at which the environment changes, the capabilities of the agent, etc.

In general, it may not be possible to achieve every goal or respond to every change in the agent's environment by the relevant deadline. In such situations, it is frequently more important to achieve some goals than others. For example, an agent should not adopt an intention of writing a research proposal, which must be submitted by 4pm on Friday if there is insufficient time to write the proposal. We therefore assume that each goal is associated with a *priority* which specifies the importance of achieving the goal. We define a *priority-maximal* set of intentions as a maximally feasible

[1]Some computer systems (for example, real-time video) utilise a stricter notion of real-time guarantees, where the precise time at which a response is produced matters [10], [13]. Hard real-time for this type of system requires a response at an exact time rather than before a deadline, and soft real-time means that the response time lies within a defined uncertainty range around the required time.

set of intentions which contains high priority intentions in preference to low priority intentions (a precise definition is given in Chapter 6.1).

In soft real time applications, occasional failure to execute an intention successfully is acceptable. We assume that the degree of "softness" of a task environment can be characterised by a *confidence level* α which is the the probability that intentions complete by their deadlines in a static environment[2]. A low value of α indicates a task environment in which failures can be tolerated. If $\alpha = 1$ and all tasks can be scheduled and never displaced by other high priority tasks then a task environment is hard real-time, i.e., all intentions must execute successfully by their deadlines.

For an agent to be real-time it must offer certain guarantees. Its cycle time and hence its reaction time (the time required for the agent to become aware of changes in its environment) must be bounded. It must commit to the "right" set of intentions, taking into account their priority and deadlines, and must schedule and execute its intentions so as to ensure their successful execution with (at least) some probability. Finally, it must update its set of intentions appropriately as its beliefs and goals change over time.

Thus we define a *real-time BDI agent* as an agent with following properties: 1) the time required to execute a single cycle of the agent interpreter (and hence the reactivity delay of the agent) is bounded by some constant; 2) the agent commits to and, with probability α, accomplishes a priority-maximal set of intentions.

[2]For simplicity, we assume that α is the same for all intentions; however, the real time guarantees we prove in Chapter 6 still hold if α is different for different events.

2.3 Summary

In this chapter, we have reviewed a generic BDI architecture and developed a notion of 'real-time' appropriate to a BDI agent. In the next two chapters we will look at existing agent architectures designed for real-time applications (Chapter 3) and outline the changes necessary for BDI architecture to implement a *real-time BDI agent* (Chapter 4).

Chapter 3

Literature Review

3.1 Introduction

In this chapter, several different agent architectures are discussed. Well-known BDI agent languages, such as PRS-CL [26], Jason [6], JIAC [20], 2APL [11], etc. provide powerful facilities for complex agent programming. Some of these architectures have several features, such as metalevel reasoning, time-outs, and plan repair rules allow agents, which are situated in dynamic real-time environments, meet real-time demands, in a limited number of applications. However these agent architectures have only *partial* or *limited* support of time-critical agent behaviour.

For example, 2APL programming language allows the user to define optional time-out parameters for the basic actions, which can be used to define the time at which an action should be considered to have failed. Such time-outs can be used to establish a bound on the maximum execution time of simple plans consisting of basic actions. 2APL also provides constructs to implement reasoning rules: planning goal rules, procedural rules and plan repair rules for plans. 2APL architecture is built on top of the Jade platform. The JADE platform and other similar architectures like Jadex,

JACK, JIAC, etc. leave the full implementation of real-time support to an agent developer i.e., the agent has to be manually programmed using the Java language.

The purpose of this chapter is to provide background information on different agent architectures which can operate in complex dynamic environments. In this chapter we discuss two types of agent architectures: those with embedded real-time control e.g., PRS and its descendants (Sections 3.2.1–3.2.3), SRTA (Section 3.2.4) and those with cooperative real-time control e.g., ROACS and SIMBA (Section 3.3.1 and 3.3.2).

Below, we describe these agent architectures in details. While all of them provide facilities for implementing complex agent systems, we have identified several features that may guarantee time-limited behaviour of a particular agent. Finally, we evaluate the ability of other well-known agent architectures to satisfy required real-time constraints in highly dynamic environments.

3.2 Embedded Real-Time Control

In this section, we review agent architectures with embedded real-time control. These architectures implement real-time capabilities using techniques and features from Artificial Intelligence (AI). Procedural Reasoning System (PRS) [15] and its descendants e.g., JAM, SPARK (Sections 3.2.1–3.2.3) are agent architectures in which the real-time control which has to be individually programmed for a particular application. In the other hand, Soft Real-Time Architecture (SRTA) and AgentSpeak(XL) (Section 3.2.4) are embedded real-time agent architectures with built-in real-time capabilities.

3.2.1 Procedural Reasoning System

The PRS (Procedural Reasoning System) was originally developed by Georgeff and Lansky [15]. The most recent version of PRS (PRS-CL) was developed by Myers [26]. In contrast to the original PRS, PRS-CL employs a representation of actions called Acts, rather than the Knowledge Areas (KAs). Act utilises as the input language for procedural knowledge in PRS-CL, however the KA representation is still used internally. The Act syntax supports certain capabilities not possible with KAs, such as required resources for the duration of the Act, properties of the Act, etc.

PRS's architecture consists of four components:

1. a database containing current *facts* (beliefs) about the world,
2. a set of current *goals*,
3. a set of *plans*, called *Acts*, and
4. an *intentions graph* (intention structure).

PRS connects these four components via an *interpreter*, which manipulates them to select and execute appropriate Acts, based on the system's facts (beliefs) and goals.

The PRS interpreter operates as follows: (1) at any particular time certain goals are posted and certain events occur that add corresponding facts to the system's database; (2) these changes in the system's goals and facts trigger a set of applicable Acts; (3) one or more of these applicable Acts will be chosen and placed on the intention graph; (4) PRS selects a task (intention) from the set of eligible intentions and (5) executes one step of that task. The result of the execution is either (6) the performance of a primitive action in the world, (7) the establishment of a new subgoal

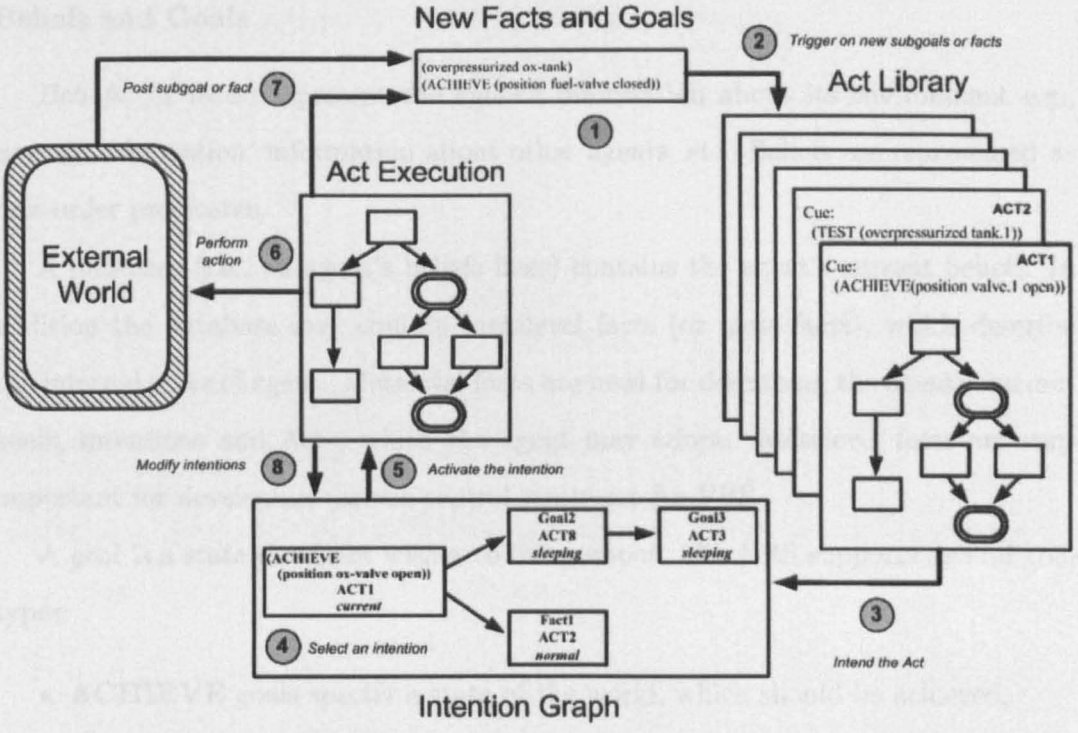

Figure 3.1: The PRS Interpreter [26]

or the conclusion of some new fact, or (8) a modification to the intention graph itself. After that the interpreter begins a new cycle.

PRS agents may operate in highly dynamic real-time environments. PRS contains several tools to support the development of real-time agent-based applications. PRS supports a priority differentiation of intentions that allows the agent to perform more important tasks first. In addition, PRS supports metalevel reasoning and planning [14], and allows the specification of time windows on individual Act activity [27] i.e., earliest and latest start times, earliest and latest finish times, and minimum and maximum durations. All of these features are discussed in turn below.

Beliefs and Goals

Beliefs (or facts) represent the agent's information about its environment e.g., sensory information, information about other agents, etc. Beliefs are represented as first-order predicates.

A database (i.e., an agent's beliefs base) contains the agent's current beliefs. In addition the database may contain metalevel facts (or meta-facts), which describe the internal state of agent. Metalevel facts are used for describing the agent's current goals, intentions and Acts, which the agent may adopt. Metalevel facts are very important for developing various control strategies for PRS.

A *goal* is a state the agent wishes to bring about. The PRS supports several goal types:

- **ACHIEVE** goals specify a state of the world, which should be achieved;
- **ACHIEVE-BY** goals are similar to achieve goals, but which should be accomplished by a restricted set of Acts;
- **TEST** goals specify a formula or formulae that must be true in the current database;
- **USE-RESOURCE** goals specify a set of resources (i.e., a physical or virtual component of limited availability within an agent) required by an Act, which must be available for the Act to be executed;
- **WAIT-UNTIL** goals make the agent waits until a given condition is true;
- **REQUIRE-UNTIL** goals specify that the agent should maintain a goal until certain condition is satisfied;
- **CONCLUDE** goals add information to the database;

- **RETRACT** goals retract information from the database.

The developer can also define meta-level goals which characterise the internal behaviour of an agent.

Acts

Acts (i.e., Plans) are sequences of actions, subgoals and complex syntax constructs that an agent can perform to achieve one or more of its intentions.

Each PRS-CL Act consists of two main parts:

1. a plot, which describes steps of the procedure to achieve an objective in a given situation (represented as plan schema or graph) and

2. an environment, which includes the triggering conditions which specify when the Act can be applied and information about the Act e.g., Name of the Act, Comments and Properties of Act.

The plot of the Act is a directed graph whose nodes represent actions to be executed, and whose arcs define a partial temporal order of execution. The plot has a single start node, but may have multiple terminal nodes, which have no outgoing arcs.

Multiple branches from a single node represent nondeterministic choice among several activities for achieving a goal. The PRS interpreter executes individual successor nodes of the split node and ignores all other branches from this node. If the goal expression on its choice is satisfiable, then it will continue executing that branch. Overwise, it will try to satisfy another node. If the split node has no successors that can be satisfied, then the split node is said to fail. Loops are represented by connecting the outgoing arc of one node to an ancestor node in the graph. Figure 3.2 presents

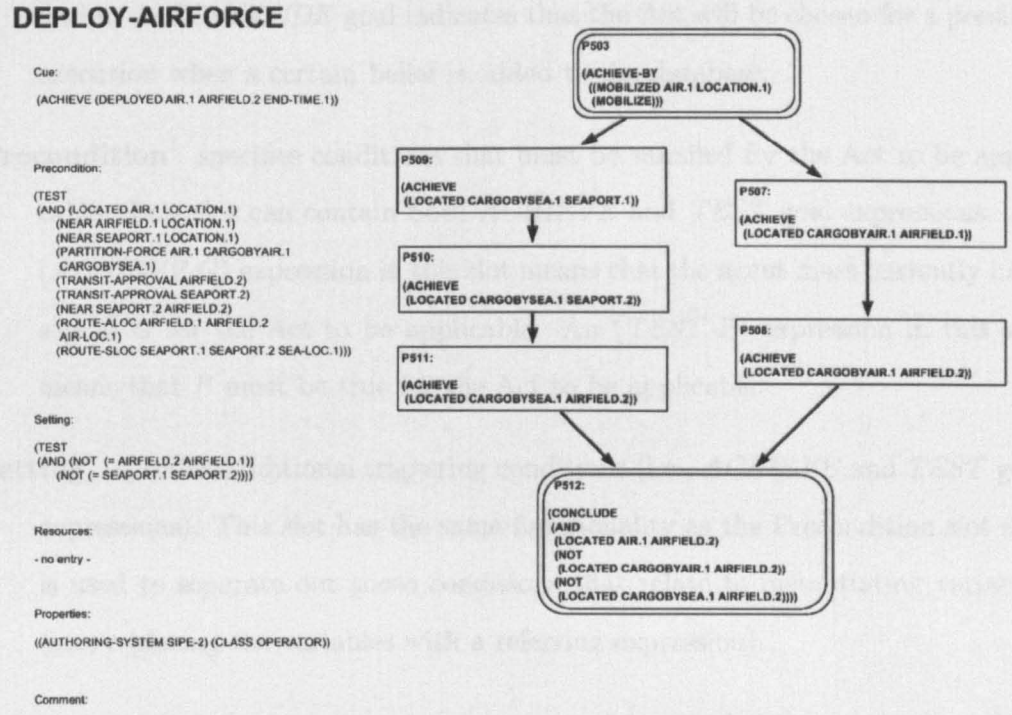

Figure 3.2: An Example of an Act plot [27]

an example Act for deploying an air force to a particular location. The environment conditions are displayed on the left side of the screen and the plot nodes on the right side.

The environment of the Act consists of six slots:

Name is an unique symbolic identifier of the Act.

Comment is a string for notes.

Cue indicates the purpose of the Act and is used to index and choose Acts for possible execution. The Cue can contain either an *ACHIEVE*, *TEST* or *CONCLUDE* goal. An *ACHIEVE* goal means that the Act is used to achieve some condition. A *TEST* goal means that the Act is used to actively test some con-

dition. A *CONCLUDE* goal indicates that the Act will be chosen for a possible execution when a certain belief is added to the database.

Precondition specifies conditions that must be satisfied for the Act to be applicable. This slot can contain both *ACHIEVE* and *TEST* goal expressions. An (*ACHIEVE G*) expression in this slot means that the agent must currently have a goal G for the Act to be applicable. An (*TEST P*) expression in this slot means that P must be true for the Act to be applicable.

Setting specifies additional triggering conditions (i.e., *ACHIEVE* and *TEST* goal expressions). This slot has the same functionality as the Precondition slot and is used to separate out those conditions that relate to instantiating variables (i.e., replacing the variables with a referring expression).

Resources specifies resources that are required for the duration of the Act. Only *USE-RESOURCE* goals can be used for this slot. A resource in PRS is a non-sharable object. Resources will be unavailable for use by other Acts until execution of the Act is finished.

Properties consists of a list of property/value pairs. The list of *Properties* is usually used to reorder the intention graph.

The `Properties` slot is very important for time-critical applications. It allows the agent programmer to specify various properties of the Act. An optional property of the Act is its *priority*. The priority is a value, which describes the relative importance of the Act. This property is used for meta-reasoning about choosing among multiple applicable Acts. For example, a user may implement a meta-Act, which intends applicable Acts with the highest priority for each goal; otherwise, an Act is selected randomly.

In addition, PRS-CL [27] provides a wide range of temporal relationships between two plot nodes within an Act using both the `Time-Constraints` property in the environment `Property` slot and the `Orderings` attribute of plot nodes. User may define a relation for a plot node in terms of Allen relations (which define possible relations between time intervals) [1], such as **starts**, **overlaps**, **before**, **meets**, **during**, **finishes**, **equals**. For example a plot node has the relationship before or meets with its successor plot node.

Also Acts allow the specification of time windows on individual plot nodes. The `Time-Window` attribute provides absolute temporal constraints on the execution time of the node, in terms of the earliest and latest start times (**start0** and **start1**), earliest and latest end times (**end0** and **end1**), and minimum and maximum durations (**min** and **max**).

This attribute for a plot node can be expressed as follows

```
(Time-Window start0 start1 end0 end1 min max).
```

The start and end times are either numbers, or one of the values **inf** (infinity), **neginf** (negative infinity), **eps** (epsilon), or **negeps** (negative epsilon). The maximum and minimum duration must be greater than zero.

These temporal reasoning properties provide powerful abstraction for an agent programmer to implement complex interactions of actions within the Act and time-bounded reactivity, which is required in highly dynamic environments. However a value of each `Ordering` or `Time-Constraints` property has to be set by the developer or a meta-procedure in each particular case.

Acts in PRS are used not only for dealing with the environment, but also can be used to manipulate the facts, goals, and intentions of a PRS agent. Such Acts are called metalevel Acts (see Section 3.2.1 for details).

Intentions

The intention structure contains all Acts that have been chosen for immediate or later execution in response to some posted *goals* or *facts*. Each intention consists of some initial *Act* together with all the "sub-Acts" being applied to satisfy the subgoals of the original *Act*, and can be viewed as a tree or a graph, where successive layers in the tree correspond to levels of subgoal within the Acts.

Information about each intention consists of (1) the purpose of the intention (goal or fact), (2) the priority of the intention, which describes their relative importance for execution stage (by default 0) and (3) the current state of the intention. The intention can have one of three possible states: 1)*Normal* means that this intention is eligible for execution; 2)*Sleeping* means that the intention is suspended and is awaiting for some activation condition; 3)*Awake* is similar to the Normal state, except that when there is more than one intention eligible for execution, the most recently awoken intention is given priority.

The set of intentions is stored in a structure called the *intention graph*. The intentions in an intention graph are partially ordered, with possibly multiple least elements, called roots. Intentions which are earlier in the ordering must be either realised or dropped (and thus disappear from the intention graph) before intentions appearing later in the ordering can be executed. This precedence ordering allows prioritized execution of intentions. The priority of an intention has to be set manually by accessing the priority slot of the intention class while intending an Act. Alternatively, an agent developer may define a metalevel procedure (i.e., meta-Act) to reorder the intention graph.

Figure 3.3 displays an ASCII representation of an intention graph, which was generated for deploying an airforce. The intention graph contains three root intentions,

```
** INTENTION GRAPH **
  *(ACHIEVE (DEPLOYED Transport-A1 London      500)) Priority: 2 State: (N)
  *(ACHIEVE (DEPLOYED Transport-A2 Belfast     500)) Priority: 1 State: (N)
  *(ACHIEVE (DEPLOYED Transport-A3 Farnborough 500)) Priority: 0 State: (N)
```

Figure 3.3: A Sample Intention Graph Summary

each with a different priority. The intention for deploy Transport-A1 in London has highest priority and will be executed first.

Metalevel reasoning

The most powerful feature of PRS is a metalevel reasoning, which can be used to handle highly dynamic environments. Metalevel and baselevel Acts have the same structure, but differ in their area of application. Metalevel Acts are used to modify the internal structures of an agent (e.g., intention structure, plan choice process etc.) using metalevel facts, goals and predefined functions.

In each execution, the agent's interpreter posts several meta-facts, such as the set of applicable acts (SOAK), the set of fact-invoked and goal-invoked applicable acts (FACT- INVOKED -KAS and GOAL-INVOKED -KAS), failed goals (FAILED-GOAL), etc. These facts start a self-reflection cycle (or applicability testing cycle), which includes the following steps: (1) trigger one or more applicable meta-Acts, and (2) conclude a new meta-fact about the set of Acts applicable and possible other meta-facts about world. The system continuously reflects on itself until there are no new applicable Acts. When the system reaches this state, one or more applicable Acts from those applicable at the previous reflection cycle will be chosen randomly. The agent developer can therefore define very complex agent behaviour with several levels of reasoning. Typically, metalevel reasoning is used to:

1. Reason about *multiple applicable Acts* in a single interpreter cycle. Using meta-Acts, a developer can change default choice process and intend Acts for a goals based on the Act priority, time availability, cost measure or other properties in order to provide best outcome.

2. *Prioritize* an agent's intentions. Assigning priority values to an agent's intentions allows execution of agent's tasks based on the priority order.

3. Implement *scheduling strategies.* PRS metalevel procedures can also reorder the intention graph based on intention priorities, plan properties etc. to ensure reactivity.

4. *Failure handling.* Using metalevel capabilities, a failed task can be resubmitted or killed if it is no longer appropriate.

New intentions without metalevel reasoning are inserted as a new root intention and removed when completed. However, metalevel Acts may change this ordering based on the task priorities, time availability and so on. These metalevel capabilities can be used to provide real-time guarantees for an agent operating in a dynamic environment, but have to be programmed individually for each particular application.

The ideas underlying PRS have been used as a basis for the implementation of various BDI agent languages. One well-known direct descendant of PRS is AgentSpeak(L), which can be viewed as a simplified variant of PRS [28]. AgentSpeak(L) and other more recent systems, such as JAM [18], SPARK [25], ARTS [36] are briefly described below.

3.2.2 JAM

JAM [18] is a BDI agent architecture and a Java implementation of PRS, which also combines ideas from the Structured Circuit Semantics (SCS) architecture [21] and Act plan interlingua [27, 41]. Like PRS, the JAM agent architecture provides procedural knowledge representation and a metalevel.

A JAM agent is composed of five main components: a world model, a plan library, an interpreter and an observer. The world model is a database, which contains the agent's current beliefs. Beliefs are represented as first-order predicates.

JAM supports three types of goals: `ACHIEVE`, `PERFORM`, and `MAINTAIN`. An `ACHIEVE` goal specifies a desire to achieve a certain state. For `ACHIEVE` goals, the interpreter checks whether the goal has been already achieved. In contrast to `ACHIEVE` goals, `PERFORM` goals are not checked to see if the goal has been achieved already. A `MAINTAIN` goal is a goal that must be posted again if it is accomplished. In addition, the agent programmer may specify *utility* of a goal, which is either a fixed numeric value or specified as an utility function.

JAM plans define a procedural specification for reacting to a goal or to changes in the environment. JAM supports both goal-driven and belief-driven plans. JAM plan applicability is constrained to two types of conditions: a *precondition*, which specifies conditions that must be true before plan execution, and a *context*, conditions that must be true before and during plan execution. The plan body can contain simple actions (e.g., user-defined primitive actions) and complex constructs (e.g., loops, deterministic and non-deterministic choice). JAM also supports constructs such as `DO ...WHILE` (loop with postcondition), `WHILE` (loop with precondition) `OR` (do any in order), `AND` (do all in order), `DO_ALL` (do all randomly), `DO_ANY` (do any randomly), `WHEN` (conditional execution) etc.

In addition, each plan may include an explicitly or implicitly defined utility, which is used for reasoning about alternative plans, i.e., to select the best alternative plan in a given situation. The plan utility is either a fixed numeric value or a utility function. The utility of an intention is a combination of the goal's utility and the utility of an instantiated plan for this goal. The JAM agent dynamically switches between its intentions as utilities change, and it always executes the intention with highest utility. The default utility value for an intention is 0.0. However plan and goal utility values are independent from each other, so it can be very hard to predict the execution order of intentions.

Another difference from PRS is the availability of an observer procedure. The observer is an optional declarative procedure, which is executed between each action in a plan. The observer is implemented as a plan body. The observer procedure allows an agent developer to control the execution of actions. However, the observer procedure cannot be complex, because it is executed very frequently.

Also JAM does not support time-constraints and ordering properties and provides limited support for metalevel reasoning: a developer may only define a metalevel procedure to reason about multiple applicable plans.

3.2.3 SPARK

SPARK [25] is a multi-agent architecture developed at the Artificial Intelligence Center of SRI International. SPARK is a Belief Desire Intention (BDI) Agent framework, which incorporates theoretical ideas from the PRS family of agent languages. It supports reasoning techniques for a procedure validation, automated synthesis, and repair. SPARK includes an expressive well-defined procedure language with a wide range of control structures, introspection capabilities, a metalevel control and

advisability techniques that support high-level user directability. It also extends PRS in including a well-defined failure mechanism.

A SPARK agent consists of a knowledge base (i.e., belief base), a library of procedures (i.e., plan library), a set of intentions and an executor (i.e., interpreter). In comparison to PRS-CL, SPARK supports two types of agent tasks (i.e., goals): basic (Table 3.1) and compound (Table 3.2).

[noop:]	Do nothing.
[fail:]	Fail.
[conclude: ϕ]	Add fact ϕ to the knowledge base.
[retract: ϕ]	Remove facts matching ϕ from the knowledge base.
[do: α]	Perform the action α.
[achieve: ϕ]	Attempt to make ϕ true.

Table 3.1: SPARK Basic Task Expressions [25]

[seq: τ_1 τ_2]	Execute τ_1 and then τ_2.
[parallel: τ_1 τ_2]	Execute τ_1 and τ_2 in parallel.
[if: ϕ τ_1 τ_2]	If ϕ is true, execute τ_1 otherwise execute τ_2.
[try: τ τ_1 τ_2]	If τ succeeds, execute τ_1 otherwise execute τ_2.
[wait: ϕ τ]	Wait until ϕ is true, then execute τ.
[while: ϕ τ_1 τ_2]	Repeat τ_1 until ϕ has no solution then execute τ_2.

Table 3.2: SPARK Compound Task Expressions [25]

SPARK supports metalevel control of the agent execution. Metalevel reasoning can be used for 1) logging of information about the agent's state and progress of execution; 2) handling the failure of tasks; 3) customizing the default procedure selection mechanism. SPARK also provides a set of predicates and actions to allow

access an agent's intention structure. These functions and actions can be used to re-order the intention structure and to implement different scheduling strategies.

A unique feature of SPARK is support for agent directability through user specified guidance. The guidance is a set of declarative polices and it is used to restrict or enable agent activities. SPARK allows an agent programmer to define two types of guidance: *strategy preference*, i.e., recommendations on how an agent should accomplish tasks, and *adjustable autonomy*, i.e., recommendations that allow the user to control the degree of agent autonomy. The guidance can be dynamically asserted, retracted and modified depending on the agent programmer preferences. Guidance provides the agent programmer with a way of adjusting agent behaviour for different application including real-time applications. It is implemented using metalevel procedures, which test properties associated with tasks, procedures, and actions.

3.2.4 Soft Real-Time Architecture and AgentSpeak(XL)

Perhaps the work most similar to the topic of this thesis are architectures such as the Soft Real-Time Agent Architecture (SRTA) [37] and AgentSpeak(XL) [3]. In comparison to PRS, these architectures provide built-in real-time capabilities and use the TÆMS (Task Analysis, Environment Modelling, and Simulation) framework [12] together with Design-To-Criteria scheduling [38] to schedule intentions.

TÆMS model and DTC scheduler

TÆMS provides a high-level framework for specifying the expected quality, cost and duration of methods (actions) and relationships between tasks (plans). Methods and tasks can have deadlines, and TÆMS assumes the availability of probability distributions over expected execution times (and quality and costs).

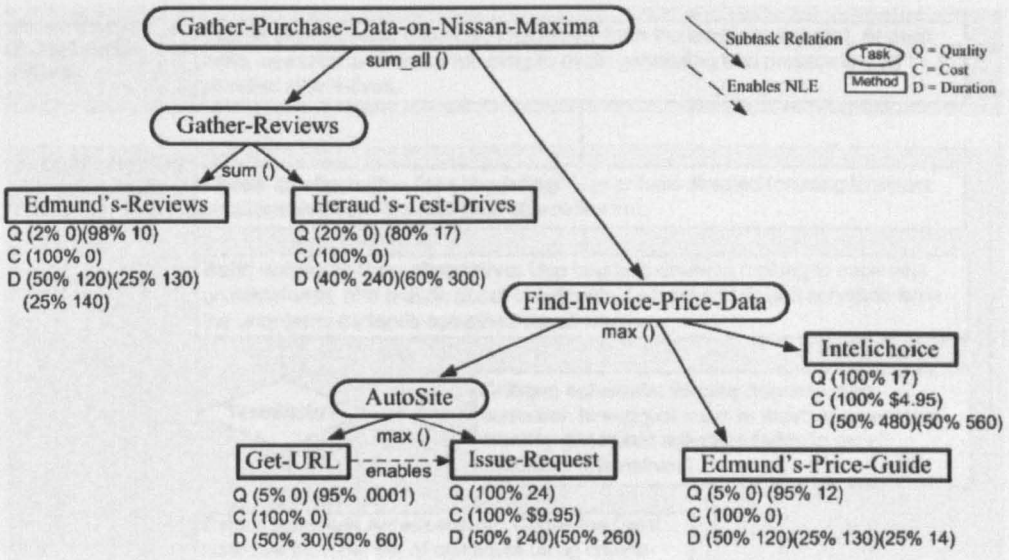

Figure 3.4: TÆMS Task Structure example [38]

Design-to-Criteria (DTC) decides which tasks to perform, how to perform them, and the order in which they should be performed, to satisfy hard constraints (e.g., deadlines) and to maximise the agent's objective function (Quality Accumulation Function) using a criteria specification metaphor, called importance sliders. These importance sliders allow the relative importance of quality, cost, and duration to be defined in terms of raw goodness, thresholds and limits, certainty, and certainty thresholds parameters. Importance sliders are set for each application.

TÆMS allows the specification of complex interactions between tasks, and DTC can produce schedules that allow interleaved or parallel execution of tasks. However, the view of 'real-time' used in these systems is different from that taken in Chapter 2. Deadlines are not hard (tasks still have value after their deadline) and no attempt is made to offer probabilistic guarantees regarding the successful execution of tasks.

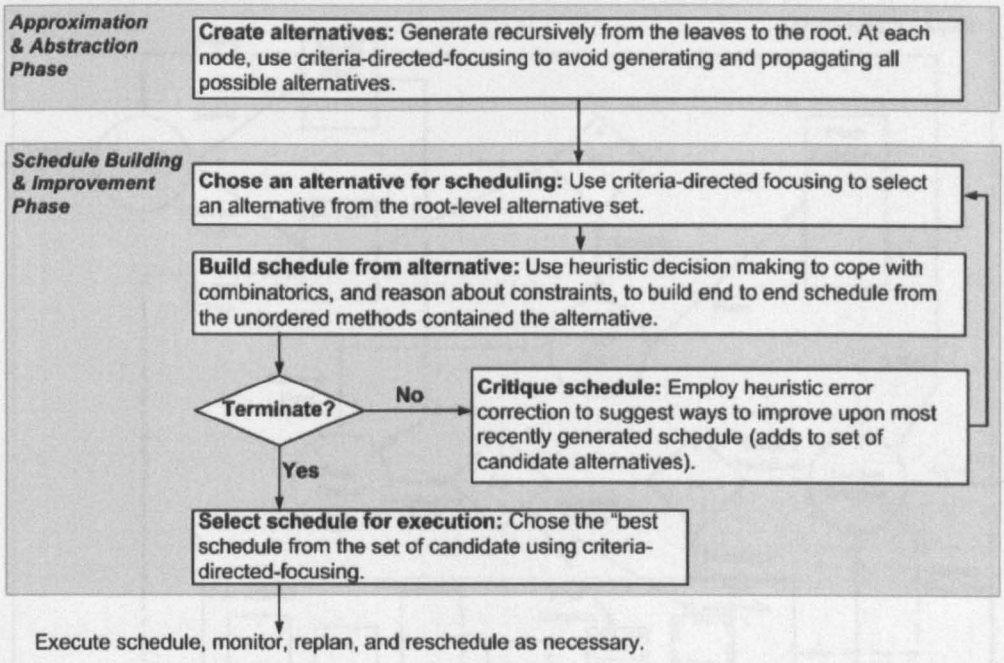

Figure 3.5: Design-To-Criteria task scheduling [38]

AgentSpeak(L)

AgentSpeak(L) is an abstract agent programming language introduced by Rao in [28]. The agent language is based on restricted first-order logic with events and actions.

The AgentSpeak(L) architecture consists of five main components: a belief base, a set of events, a plan library, an intention structure, and an interpreter (see Figure 3.6).

AgentSpeak(L) differs from PRS in following significant respects:

- **Syntax.** In contrast to PRS AgentSpeak(L) has a simpler syntax. It distinguishes two types of goals: *achievement* (i.e., describes a state that an agent wants to achieve) and *test* (i.e., determines whether the associated predicate is true). AgentSpeak(L) plans consists of only primitive actions and subgoals.

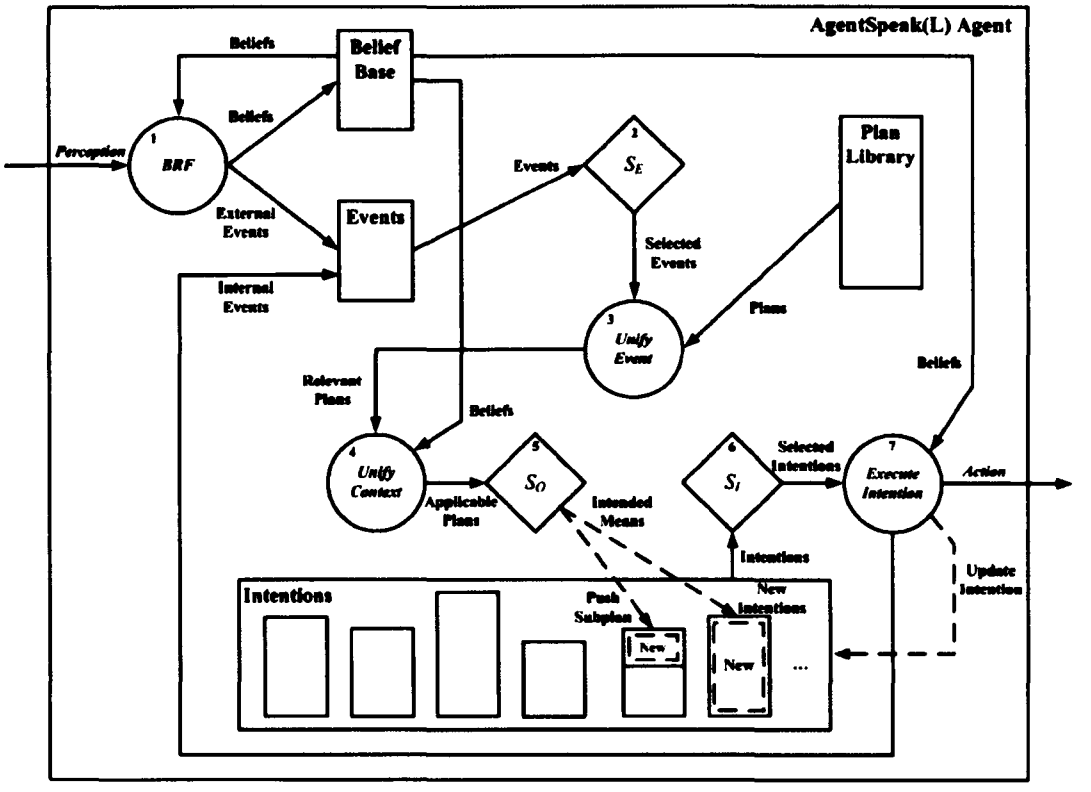

Figure 3.6: AgentSpeak(L) Interpreter [24]

- **Real-time capabilities.** AgentSpeak(L) does not explicitly support metalevel reasoning (it can be implemented through selection functions). The absence of meta-level reasoning and the ability to assign priority and deadlines for agent's events make it difficult to develop of AgentSpeak(L) agents for real-time applications.

However, its simple and well-defined structure makes it possible to develop extensions.

AgentSpeak(XL)

AgentSpeak(XL) [3] is an extension of the well known AgentSpeak(L) language. The AgentSpeak(XL) framework extends AgentSpeak(L) by incorporating the TÆMS (Task Analysis, Environment Modelling, and Simulation) framework [12] and DTC (Design-To-Criteria) scheduling [38] to schedule intentions.

SRTA: Soft Real-Time Agent Architecture

SRTA (Soft Real-Time Agent Architecture) [37] provides facilities to support the to develop agent systems development of soft real-time requirements. This architecture consists of a Problem Solver, a TÆMS library, DTC and Partial Order Scheduler, Execution and Learning modules (see Figure 3.7).

The SRTA scheduler includes conflict resolution, task merging and resource modelling modules. The conflict resolution module reasons about mutually exclusive tasks and determines the best way to resolve conflicts. The task resolution module allows the scheduler to combine several tasks within an existing schedule. Finally, the conflict resolution module is used to ensure that resource constraints are satisfied for each agent task. A Problem Solver is responsible for translating high-level goals into TÆMS tasks and handling task failures. The learning component of SRTA is used to monitor task execution and update the task template library when new trends are observed. It should be noted that although DTC can be used in an 'anytime' fashion, neither SRTA or AgentSpeak(XL) execute in bounded time.

3.3 Cooperative Real-Time Control

Agent architectures based on the cooperative real-time control approach consist of both intelligent and real-time control parts. This type of architecture is often called

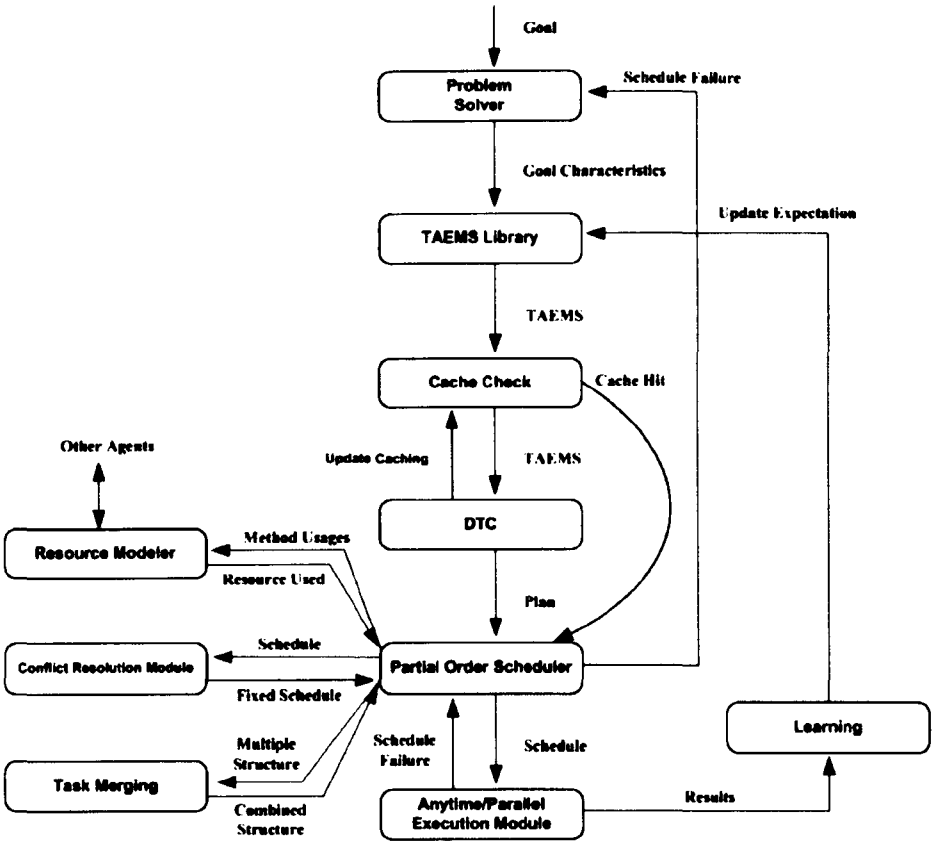

Figure 3.7: Soft Real-Time Agent Architecture [37]

hybrid. A wide range of designs have been proposed, differing only in complexity of the processing available on AI and real-time subsystems, and the communication between these subsystems. Hybrid agent architectures such as ROACS [16] and SIMBA [9] consist of an AI subsystem and a low-level control subsystem connected by a communication interface. Such systems attempt to improve responsiveness by separating the 'real-time' aspects of the architecture from the high-level control.

3.3.1 ROACS

ROACS (Real-time Open-Architecture Control System) [16] is a hybrid agent architecture, developed for the flexible control of industrial robots. ROACS is capable of handling tasks with uncertainties. The communication between the AI and real-time subsystems is based on a client-server communication model. ROACS was implemented as vertical hierarchy control system with information data feedback at every level. Figure 3.8 shows the ROACS architecture.

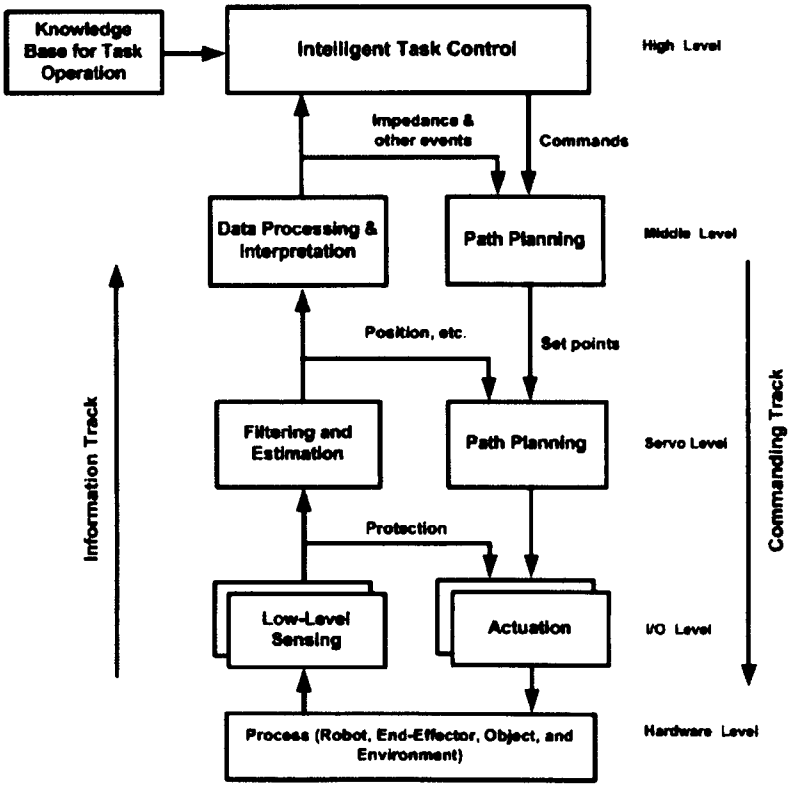

Figure 3.8: ROACS architecture [16]

The intelligent subsystem is responsible for deciding which task to perform based on informational feedback and agent knowledge. The real-time subsystem in turn executes these chosen tasks. ROACS tasks are executed in priority order. ROACS

also allows specification of motion restrictions for a task to restrict default agent motions, and specification of a task set to enforce execution order of some sequential motion commands.

However, the ROACS architecture has several drawbacks: it does not support timely execution of tasks and is highly dependent on the hardware configuration and communication between intelligent and real-time subsystems.

3.3.2 SIMBA

SIMBA [9] is a multi-agent platform for the development of real-time systems. SIMBA incorporates a set of real-time ARTIS agents [7], and a special Manager Platform agent (MPA). The SIMBA architecture is shown in Figure 3.9. An ARTIS agent is an extension of the blackboard model[1], which is adapted to work in real-time environments. The SIMBA architecture can also include case-based planning BDI (CBP-BDI) agents, which are specialized in generating optimum plans for ARTIS agents. Agents in the SIMBA platform communicate through UDP/IP protocol.

The intelligent part of an ARTIS agent is made up of a set of sensors and effectors, which provide the interaction between the agent and its environment, a set of beliefs, and set of behaviours. The behaviour of the ARTIS agent is determined by a set of in-agents, which periodically perform a specific task. Each in-agent solves a particular subproblem, which is a part of the entire problem. An in-agent may use results obtained by other in-agents. ARTIS distinguishes two types of in-agents: critical, which requires time-limited response, and non-critical, which allows anytime response. It is possible to define a deadline and a period. An in-agent consists of reflex and real-

[1]The blackboard architectural model, where a common knowledge base, the "blackboard", is iteratively updated by a diverse group of specialist knowledge sources, starting with a problem specification and ending with a solution.

time deliberative layers. The reflex layer produces a minimum quality response and real-time deliberative layer attempts to improve this response. Non-critical agents only have the real-time deliberative layer.

A control part in ARTIS agent provides real-time execution of in-agents in a particular hardware and allows in-agents to satisfy temporal requirements for in-agents. Different execution criteria for two in-agent layers must be specified by the developer.

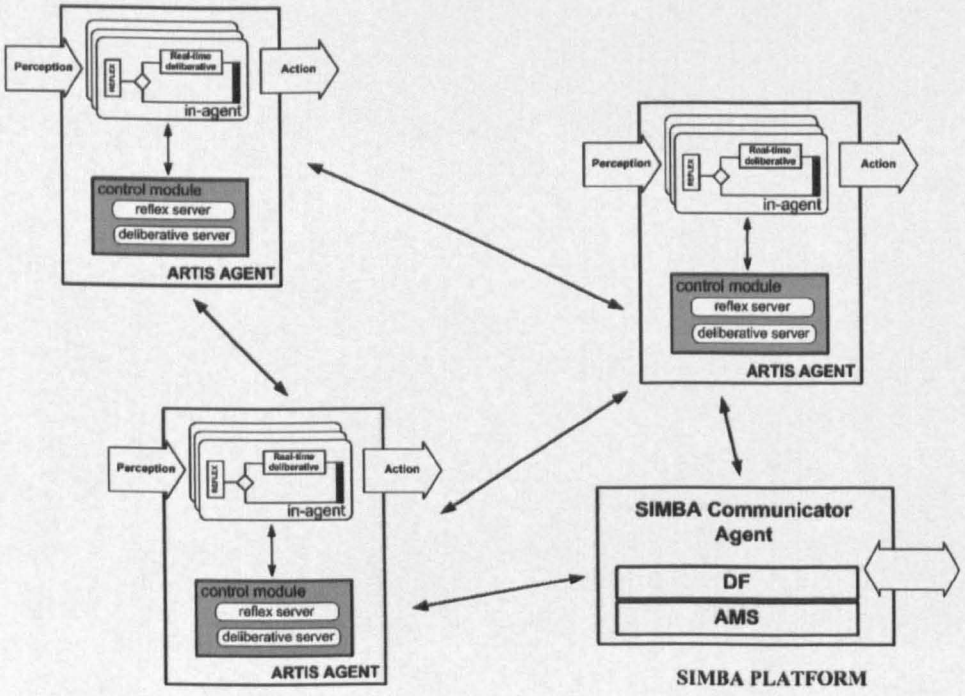

Figure 3.9: SIMBA architecture [9]

ARTIS agents together with CBP-BDI agents, which generate plans for ARTIS agent, allow the development of agent systems for real-time applications. However the SIMBA platform does not support real-time communication between agents or communication within ARTIS agents i.e., it is not guaranteed to receive the packets reliably and on time.

It should be noted that SIMBA agents on the one hand contain two subsystems: intelligent subsystem and control module, as ROACS agents, but on the other hand both subsystems are capable of providing real-time behaviour of the agent. Thus, the SIMBA platform can be categorised as an agent architecture with both cooperative and embedded real-time control.

While such agent architectures with cooperative approach to real-time guarantees can simplify the development of agents for real-time environments, they provide limited high-level support for managing the timely execution of tasks as an agent developer has to deal with the problem of real-time communication and synchronization between an intelligent and a control parts of an agent.

3.4 Summary

In this chapter, a number of agent architectures for a time critical environment have been introduced. These architectures were divided in two categories: architectures with embedded real-time control and architectures with cooperative real-time control. The embedded real-time agent architectures are architectures with built-in real-time capabilities e.g., SRTA, AgentSpeak(XL), and architectures which has to be individually programmed e.g., PRS. These architectures provide a powerful abstraction for developing real-time agent systems, but they either must be tuned for a particular application, or can be applied only to a limited number of applications.

Agent architectures based on the cooperative approach couple intelligent and real-time subsystems e.g., ROACS and SIMBA. However while such systems can simplify the development of agents for real-time environments, they provide limited high-level support for specifying the tasks themselves.

In the next chapter we outline the changes necessary to a BDI architecture to

implement a *real-time BDI agent.* The proposed method allows any BDI agent to meet real-time constraints.

Chapter 4

Changes to the BDI Architecture

In this chapter we explain changes that must be made to a BDI architecture to implement a *real-time BDI agent.* We assume a simple generic BDI architecture in which an agent has beliefs and goals, and selects plans (sequences of subgoals and primitive actions) in order to achieve its goals or in response to new beliefs. Once the agent has adopted a plan it becomes an intention, and at each cycle the agent executes a single step of one of its current intentions. To implement real-time BDI agents within such an architecture, two main changes are required: we must extend the BDI model to include additional information about events and plans to support real-time guarantees, and we need to change the BDI execution cycle to ensure that the agent's cycle time is bounded and that, with probability α, a priority-maximal set of intentions is successfully executed by their deadlines. We consider each of these changes in turn below.

4.1 Extending the BDI Model

As discussed in Chapter 3, in order to provide real-time guarantees, each top-level event must be associated with a *deadline* which specifies the time by which the agent should respond to an event. We assume that the deadline for an event is specified when the event is generated by a user (or another agent), and is expressed as a real time value in some appropriate units (milliseconds, minutes, hours etc.). By default, the plan selected to respond a top-level event (and its subgoals and subplans) inherit the deadline of the top-level event.

Each top-level event is also associated with a *priority* which specify the relative importance of responding to a particular event. The priority of a top-level event determines the priority of the corresponding intention. The subgoals generated by an intention inherit the priority of the intention.

Each plan is also associated with a *duration*, an estimate of the real time necessary to execute the plan. The expected execution time for an action or plan ϕ at confidence level α is given by $et(\phi, \alpha)$. We assume that execution times increase monotonically with α, i.e., in general, to have higher confidence that a plan will complete successfully, we need to allow more time for the plan to execute.

Each plan may be optionally associated with a *plan priority* or a *plan cost* which specifies the relative utility or a cost of the plan execution. The plan priority or the cost can be used to choose between multiple applicable plans for a goal or a belief. The most natural and easiest way for solving this problem is to choose the plan that takes less time than others.

4.2 Changes to the BDI Execution Cycle

We assume that the internal operations of the agent—adding or deleting a belief or goal, selecting a plan, adopting an intention, selecting an intention to execute and executing a single step of the intention– require time bounded by the size of the agent's program and its beliefs and goals. Adding or deleting a belief or goal, adopting an intention, and executing a single step of an intention can be assumed to take constant time. However selecting a plan and intention to execute are intractable in the general case, and it is necessary to approximate the choices of an unbounded agent to limit the agent's cycle time.

To bound the time necessary to select an intention to execute at the current cycle, the agent utilises a scheduling algorithm which gives preference to high-priority intentions. The set of candidate intentions are processed in order of descending priority. A candidate intention is added to the schedule if it can be inserted in the schedule in deadline order while meeting its own and all currently scheduled deadlines. If the estimated remaining execution time for the intention or any of its sub-plans is greater than the time to remaining to the deadline of the intention, the intention is dropped. This gives a non-empty priority-maximal set of intentions provided at least one intention is feasible. There is a wide range of scheduling algorithms in scheduling theory which have polynomial complexity. Finally, the interpreter selects the first intention from the computed schedule and executes one step of that intention.

4.3 ARTS

Initially the approach was used to implemented ARTS (Agent Real-Time System) [36], an implementation of a real-time BDI agent architecture. ARTS is an

agent programming framework for agents with soft real-time guarantees. The syntax and execution semantics of ARTS is based of PRS-CL and JAM, augmented with information about deadlines, priorities, and changes to the interpreter to implement time-bounded priority driven plan selection and deadline monotonic intention scheduling. ARTS is implemented in Java, and the current prototype implementation includes the core ARTS language, and implementations of some basic primitive actions. Additional user-defined primitive actions can be added using a Java API. In the interests of brevity, we do not discuss the meta-level features of ARTS.

An ARTS agent consists of five main components: a database, a goal stack, a plan library, an intention structure, and an interpreter. The database contains the agent's current beliefs (facts). The goal stack is a set of goals to be realised. The plan library contains a set of plans which can be used to achieve agent's goals or react to particular situations. The intention structure contains plans that have been chosen to achieve goals or respond to facts. The interpreter is the main component of the agent. It manipulates the agent's database, goal stack, plan library and intention structure and reasons about which plan to select based on the agent's beliefs and goals to create and execute intentions. Changes to the agent's environment or posting of new goals invokes reasoning to search for plans that might be applied to the current situation. The ARTS interpreter selects one plan from the list of applicable plans, intends and schedules it, and executes the next step of first intention in the computed schedule.

4.3.1 Facts

The database of an ARTS agent contains facts (beliefs) that represent the state of the agent and its environment. Facts may represent information about percepts,

messages from other agents, derived information, etc.

fact	::=	*ground_wff*
ground_wff	::=	*pred_name ground_term_exp** ";" | "(NOT" *ground_wff* ")" |
		"(AND" *ground_wff*$^+$ ")" | "(OR" *ground_wff*$^+$ ")"
ground_term_exp	::=	*value* | *ground_function_exp*
value	::=	integer | float | string
ground_function_exp	::=	"(" *fun_name ground_term_exp*$^+$ ")"

where *pred_name*, and *fun_name* name predicates and functions respectively.

4.3.2 Goals

ARTS distinguishes two categories of goals: top-level goals and subgoals. ARTS supports two top-level goal operators: ACHIEVE and CONCLUDE. (ARTS does not currently support maintenance goals.) An ACHIEVE goal specifies that the agent desires to achieve a particular goal state. A CONCLUDE goal inserts a certain fact into the database. Goal states and facts are specified as ground *wff*s. For top-level ACHIEVE goals a priority and deadline may also be specified.

The form of top-level goals is given by:

goal	::=	*achieve_goal* | *conclude_goal*
achieve_goal	::=	"ACHIEVE" *ground_wff* [":PRIORITY" *p*] [":DEADLINE" *d*]
		[*by* | *not_by*] ";"
conclude_goal	::=	"CONCLUDE" *ground_wff* [*by* | *not_by*] ";"
by	::=	":BY" *plan_name*$^+$
not_by	::=	":NOT_BY" *plan_name*$^+$

where p and d are non-negative integer values and *plan_name* is the name of a plan. The :PRIORITY and :DEADLINE fields of an ACHIEVE top-level goal are optional: if they

are omitted the default priority is zero and the default deadline is infinity[1].

The developer can specify one or more top-level goals for the agent as part of the agent's program using the keyword "GOALS:". For example:

```
GOALS:
  ACHIEVE PrepareLecture agents101 : PRIORITY 9 :DEADLINE 50;
  ACHIEVE HaveLunch :PRIORITY 7 :DEADLINE 40;
  ACHIEVE BorrowBook R&N :PRIORITY 2 :DEADLINE 30;
```

Subgoals are goals generated within plans. ARTS has the following subgoals operators:

ACHIEVE C	achieve condition C
CONCLUDE F	add fact F to the database
TEST C	test for the condition C
RETRACT F	retract fact F from database
WAIT C	wait until condition C is true

In contrast to top-level goals, the deadline and priority of ACHIEVE subgoals are inherited from the plan containing the subgoal.

4.3.3 Plans

Plans define a procedural specification for achieving a goal. In specifying plans we distinguish between *plan trigger variables* and *plan body variables.* Plan trigger variables are free variables appearing in the cue, precondition and context fields, while plan body variables are variables appearing in the body of the plan. Plan trigger variables must be ground when the plan is selected, while binding of plan body variables can be deferred to the execution of the corresponding plan step. The agent's plan library is introduced by the keyword "PLANS:" followed by a list of plans

[1]Tasks with a deadline of infinity will be processed after any task with a specified deadline.

of the form:

Name is an unique symbolic identifier of the plan.

Documentation is an optional field which is used to store a descriptive text string.

Cue specifies the purpose of the plan and is used to select the plan for possible execution. The **Cue** field can contain either an `ACHIEVE` or `CONCLUDE` goal. A `ACHIEVE` goal in the **Cue** field means that the plan may be used to achieve some condition, while a `CONCLUDE` goal means that the plan may be chosen for possible execution when a fact is added to the database.

Precondition specifies conditions that must be satisfied for plan to be applicable. This field is optional and can contain both `ACHIEVE` and `TEST` goal expressions. An `ACHIEVE` G precondition means that the system must currently have **G** as a goal in order for the plan to be applicable, while a `TEST` **C** precondition means that **C** must be true for the plan to be applicable.

Context defines additional conditions (i.e. `ACHIEVE` and `TEST` goal expressions) on plan execution. This field is optional and has similar functionality to the Precondition field, but in contrast to the precondition it must be satisfied before and during plan execution. As in JAM, this significantly increases the reactivity of the agent.

Body defines a sequence of simple activities, i.e., primitive actions, addition and deletion of goals and facts, and complex constructs (e.g. loops, (non)deterministic choice, etc, see below.

Priority specifies the relative utility of the plan. The plan priority is used to choose between the applicable plans for a particular goal. The priority field is optional

and allows the specification of either a constant priority or an expression which allows the calculation of the plan priority as function of variables appearing in the plan trigger. The default priority value is 0.

Deadline specifies a deadline for the plan. The deadline field is optional and allows programmer to advance the deadline inherited from the triggering goal. The deadline can be specified as a constant value or an expression which allows the calculation of the plan deadline as function of variables appearing in the plan trigger. If the specified plan deadline is earlier than the deadline for this intention it becomes the deadline for the intention during the execution of the plan (i.e., it effectively advances the deadline for this intention during the execution of the plan). If the specified deadline is later than the deadline for the intention, the plan deadline is ignored.

ARTS, like JAM, supports standard programming constructs such as `DO ... WHILE` (loop with postcondition), `WHILE` (loop with precondition), choice constructs specified by `OR` (do any in order), `AND` (do all in order), `DO_ALL` (do all randomly), `DO_ANY` (do any randomly), `WHEN` (conditional execution), and `ASSIGN` (assignment to plan body variables).

The BNF for plans is given by:

plan ::= "`PLAN:` {"*p_name* [*p_doc*] *p_cue* [*p_precond*] [*p_cont*]
p_body [*p_pr*] [*p_dl*] [*p_attr*] "}"

p_name ::= "`NAME:`" string";"

p_doc ::= "`DOCUMENTATION:`" [string]";"

p_cue ::= "`CUE:`" *p_goal_exp* ";"

p_precond ::= "`PRECONDITION:`" *p_cond** ";"

p_cont ::= "`CONTEXT:`" *p_cond** ";"

p_body ::= "BODY:" *body_elem**

p_pr ::= "PRIORITY":" *trigger_exp* ";"

p_dl ::= "DEADLINE":" *trigger_exp* ";"

body_seq ::= "{" *body_elem** "}"

body_elem ::= *activity* | *b_and* | *b_or* | *b_parallel* | *b_do_all*
| *b_do_any* | *b_do_while* | *b_while* | *b_when*

activity ::= *prim_act* | *misc_act* | *subgoal* ";"

b_and ::= "AND:" *body_seq*$^+$ ";"

b_or ::= "OR:" *body_seq*$^+$ ";"

b_parallel ::= "PARALLEL:" *body_seq*$^+$ ";"

b_do_all ::= "DO_ALL:" *body_seq* $^+$ ";"

b_do_any ::= "DO_ANY:" *body_seq*$^+$ ";"

b_do_while ::= "DO:" *body_seq* "WHILE:" *p_cond* ";"

b_while ::= "WHILE:" *p_cond body_seq* ";"

b_when ::= "WHEN:" *p_cond body_seq* ";"

p_goal_exp ::= "ACHIEVE" *wff* | "CONCLUDE" *wff*

p_cond ::= "ACHIEVE" *wff* | "TEST" *wff*

subgoal ::= *subgoal_op wff* ";"

subgoal_op ::= "ACHIEVE" | "CONCLUDE" | "TEST" | "RETRACT" | "WAIT"

prim_act ::= "EXECUTE:" *ground_function_exp* [":TIMEOUT" *ground_term_exp*]

misc_act ::= "ASSIGN:" *ground_term_exp term_exp*

wff ::= *pred_name term_exp** ";" | "(NOT" *wff* ")" |
"(AND" *wff*$^+$ ")" | "(OR" *wff*$^+$ ")"

term_exp ::= *value* | *variable* | *function_exp*

value ::= integer | float | string

variable ::= "$" *var_name*

function_exp ::= "(" *fun_name term_exp*$^+$ ")"

where *pred_name*, *fun_name* and *var_name* name predicates, functions and variables respectively.

4.3.4 Primitive Actions

Subgoal operators are implemented directly by the ARTS interpreter. Other primitive actions are implemented as Java methods. Each primitive action referenced in a plan body must have Java code which implements the necessary functionality. ARTS supports two mechanisms for defining primitive actions: writing a class which implements the `PrimitiveAction` interface, and direct invocation of methods in existing legacy Java code. Primitive actions are executed by using an `EXECUTE` action.

In contrast to PRS-CL and JAM, ARTS allows the agent programmer to specify a timeout for each primitive action by using the `TIMEOUT` keyword. The timeout specifies the maximum amount of real time required to perform the action. Actions which do not complete by their timeout are assumed to have failed. For example:

`EXECUTE` *move-to* **$x $y** : `TIMEOUT` 50

If a primitive action times out, the intention containing the action is removed from the intention structure.

4.3.5 Interpreter

The ARTS interpreter repeatedly executes the activities shown in Figure 4.1:

1. New goals are added to the goal stack and facts corresponding to `CONCLUDE` goals and external events are added to the database.

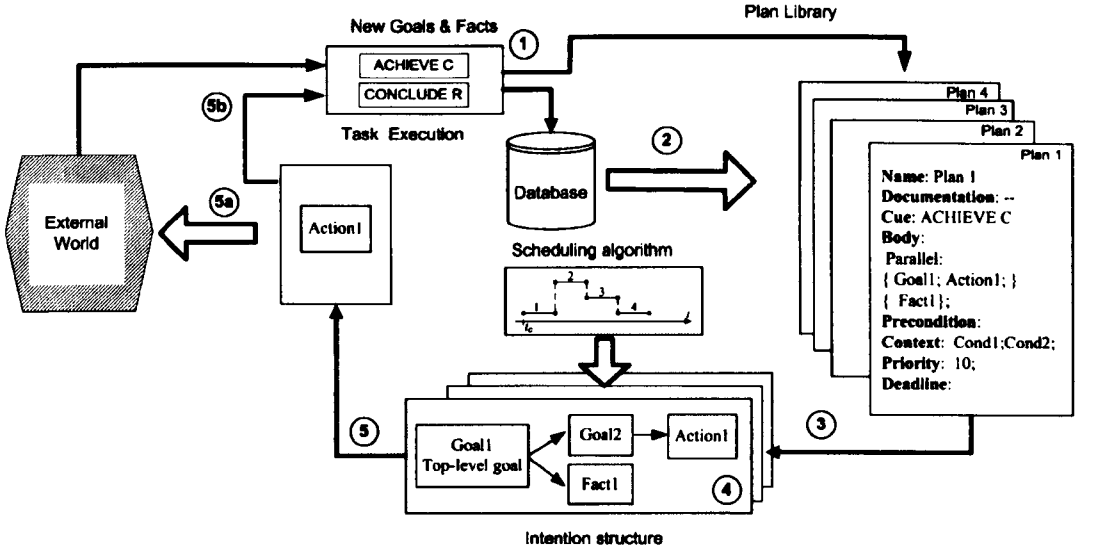

Figure 4.1: The ARTS Interpreter [36]

2. The precondition and context expressions of plans with a cue matching a goal on the goal stack are evaluated against the database to determine if the plan is applicable in the current situation. Goals and plans are matched in priority order. For ACHIEVE goals, the interpreter checks to see whether the goal has already been accomplished before trying to invoke a plan.

3. The resulting set of applicable plans are added to the set of candidate intentions.

4. Intentions are scheduled according to their deadline and priority value as described in Section 4.2. Intentions which are not schedulable, i.e., their minimum remaining execution time is greater than the time remaining to their deadline, are either dropped or have their priority reduced to zero[2].

5. Finally, the interpreter selects the first intention from the computed schedule and executes the one step of that intention. The result of the execution can be

[2]This choice is currently determined by a global flag, rather than per goal.

(5a) execution of a primitive action or (5b) the posting of a new subgoal or the conclusion of some new fact.

If no new beliefs and goals were added at the current cycle, steps 1–4 can be skipped.

4.3.6 Conclusion

ARTS is intended for the development of agent-based applications such as robot control or stock trading, where decisions must be made in a timely manner. ARTS is influenced by the PRS family architectures, such as PRS-CL and JAM. However, unlike previous PRS-like architectures, ARTS includes a duration estimation algorithm, priority driven plan selection and a deadline monotonic intention scheduling algorithm. These features enable an ARTS agent to produce an intention schedule which achieves a priority-maximal set of goals by their deadlines. While the resulting schedule may not contain the greatest number of high-priority tasks, it is computable in bounded time, and we believe that the kind of "optimistic bounded rationality" implemented by the ARTS architecture provides a simple, predictable framework for agent developers, facilitating the development of agents which can execute tasks to deadlines while providing timely responses to events in a dynamic environment.

However, complex agent behaviour and expressive syntax constructs, provided by this framework, require a lot of additional information about an agent environment and do not admit formal definitions or proofs of real-time properties.

4.4 Summary

In this chapter we proposed an approach that allows any BDI agent architecture to operate on real-time environment. In comparison to metalevel control used in PRS-like agent systems, the proposed approach provides a simple way for the development

of real-time BDI agents without significant additional adjustment. In the next chapter, we present a new simple and predictable agent framework AgentSpeak(RT) for the development of soft real-time agents.

Chapter 5

AgentSpeak(RT): A Real-Time Agent Programming Language

In this chapter we present AgentSpeak(RT), a programming language for real-time BDI agents. AgentSpeak(RT) extends AgentSpeak(L) with deadlines and priorities, and, given the estimated execution time of plans, schedules intentions so as to achieve a priority-maximal set of intentions by their deadlines with a specified level of confidence. We also present the syntax of AgentSpeak(RT) and describe the execution cycle of the AgentSpeak(RT) architecture. Finally we show an example AgentSpeak(RT) agent program for an intelligent control of a nuclear power plant.

5.1 Introduction

AgentSpeak(RT) is a framework for developing soft real-time agents which can perform complex tasks in real-time environments in which external events (goals and changes in the agent's beliefs about its environment) may be associated with a deadline and/or a priority. The agent responds to events by adopting and executing

intentions. A developer can specify a required level of confidence for the successful execution of intentions in terms of a probability, α, and the agent schedules its intentions so as to ensure that the probability that intentions complete by their deadlines is at least α[1]. If not all intentions can be executed with the required level of confidence, the agent favours intentions triggered by high priority events.

The syntax and semantics of AgentSpeak(RT) is based on AgentSpeak(L) [28]. The current version of AgentSpeak(RT) is implemented in Java, and the current prototype implementation includes the core language described below and implementations of some basic external actions. Additional user-defined actions can be added using a Java API (for more details about AgentSpeak(RT) implementation see Appendix A).

AgentSpeak(RT) contains variables, constants, function symbols, predicate symbols, action symbols, connectives and punctuation symbols. In addition to the standard logical connectives `&`, `|` and `not`, we use $-$ and $+$ (for belief addition and removal events), `!` (for achievement goals), `?` (for test goals), `;` (for sequential composition) and `<-` (for plans). We assume standard definitions of terms, literals, ground literals, and free and bound occurrences of variables.

To illustrate the syntax of AgentSpeak(RT) we use a simple running example of an agent which removes litter from a parking lot. Each evening, the agent is given a set of goals to achieve, each of which specifies the removal of a particular item of litter from particular parking space. In addition, the agent may detect additional litter while moving around the lot. There is a deadline for the removal of litter e.g., before the barrier is opened in the morning (we assume the agent can't cope with parking cars), and it is more important to remove some types of litter (e.g., broken

[1]For simplicity, we assume that α is the same for all intentions; however the real time guarantees we prove in Chapter 6 still hold if α is different for different events.

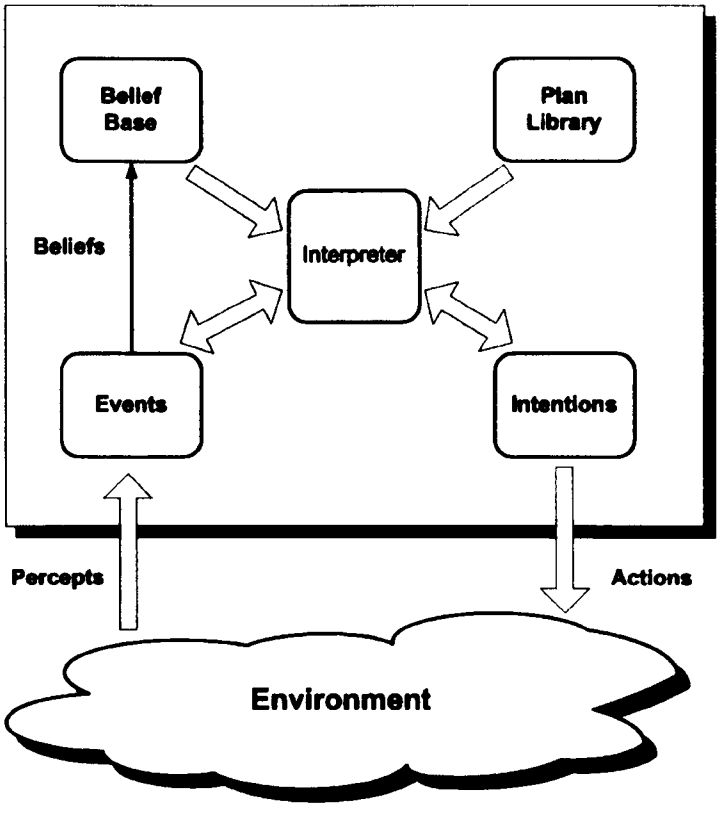

Figure 5.1: An AgentSpeak(RT) agent

glass) than others (e.g., paper).

The AgentSpeak(RT) architecture (see Figure 5.1) consists of five main components: a belief base, a set of events, a plan library, an intention structure, and an interpreter. We describe each of these in turn below.

5.2 Beliefs and Goals

The belief base contains beliefs that represent the state of the agent and its environment, e.g., sensory input, information about other agents, etc. Beliefs are represented as ground literals or conjunctions of ground literals. For example, the

agent may believe that it is in space1 and there is some litter in space2:

```
at(robot,space1)
litter(paper,space2)
```

A belief atom or its negation is referred to as a *belief literal.* A ground belief atom is called a *base belief*, and the agent's belief base is a conjunction of base beliefs.

A goal is a state the agent wishes to bring about or a query to be evaluated. An achievement goal, written $!g(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms, and g is a predicate, specifies that the agent wishes to achieve a state in which $g(t_1, \ldots, t_n)$ is a true belief. A test goal, written $?g(t_1, \ldots, t_n)$, specifies that the agent wishes to determine if $g(t_1, \ldots, t_n)$ is a true belief. For example, the goals

```
!remove(paper,space2)
?parked(X,space2)
```

indicate that the agent wants to remove the paper in space2, and determine if there is a car parked in space2.

As in Prolog, constants are written in lower case and variables in upper case, and all negations must be ground when evaluated. AgentSpeak(RT) currently supports string, integer and floating point constants.

Table 5.1 shows the BNF of goals and beliefs for the AgentSpeak(RT) language, where **p** and **f** are respectively a predicate and a functor symbols; **VAR** is a variable name. We will describe a BNF for AgentSpeak(RT) plans below in the Table 5.3.

5.3 Events

Changes in the agent's beliefs or the acquisition of new achievement goals give rise to events. An addition event, denoted by +, indicates the addition of a belief

goal	::=	"!" *literal* | "?" *literal*
belief	::=	*literal*
belief-base	::=	(*belief* ".")*
literal	::=	["not"] *atomic-formula*
atomic-formula	::=	**p** ["(" *term-list* ")"]
term-list	::=	*term* ("," *term*)*
term	::=	*constant* | *variable* | *function*
constant	::=	*number* | **string**
number	::=	**integer** | **float**
variable	::=	**VAR**
function	::=	**f** "(" *term-list* ")"

Table 5.1: BNF for AgentSpeak(RT) beliefs and goals

or an achievement goal. A deletion event, denoted by $-$, indicates the retraction of a belief[2]. We distinguish between internal and external events. An external event is one originating outside the agent while internal events result from the execution of the agent's program.

Notation	Description
$+$*literal*[*deadline*, *priority*]	Belief addition
$-$*literal*[*deadline*, *priority*]	Belief deletion
$+!$*literal*[*deadline*, *priority*]	Achieve-goal addition

Table 5.2: Types of events.

As in AgentSpeak(L) all belief change events are external (originating in the agent's environment), while goal change events may be external (goals originating

[2]In the interests of consistency with the original AgentSpeak(L) [28] semantics, we do not consider goal deletion events.

from a user) or internal (subgoals generated by the agent's program in response to an external event).

To allow the specification of real-time tasks, external events may optionally specify a deadline and a priority. A *deadline* specifies the time by which a goal should be achieved or the agent should respond to a change in its beliefs expressed as a real time value in some appropriate units, e.g, a user may specify a deadline for a goal as "4pm on Friday". Deadlines in AgentSpeak(RT) are hard – it is assumed that there is no value in achieving a goal or responding to a belief change after the deadline has passed. A *priority* (a non-negative integer value, with larger values taken to indicate higher priority) specifies the relative importance of achieving the goal or responding to a belief change. For example, the events

```
+!remove(paper,space2)[8am,10]
+litter(glass,space1)[8am,20]
```

indicates the acquisition of a goal to remove some paper from space2 with deadline 8am and priority 10, and a new belief that there is broken glass in space1; note that responding to the broken glass has higher priority than the goal to remove paper from space1. By default, the deadline is equal to infinity and the priority is equal to zero.

5.4 Plans

Plans specify sequences of actions and subgoals an agent can use to achieve its goals or respond to changes in its beliefs.

Each AgentSpeak(RT) plan specification consists of two parts:

1. a *plan*, which contains the name of the plan, a triggering condition, a context, and the body of the plan (see Section 5.4.1);

2. an *execution time profile*, which specifies the expected execution time of the plan at confidence level α. The execution time profile is specified in a separate file, i.e., it is not part of the syntax of a plan (see Section 5.4.3).

5.4.1 Plan

A plan consists of four parts: the *plan name*, the *triggering event*, the *belief context*, and the *body*. The head of a plan consists of a name which uniquely identifies the plan, the triggering event which specifies the kind of event the plan can be used to respond to i.e., its invocation condition, and the belief context which specifies the beliefs that must be true for the plan to be applicable.

Name. The name is a string that uniquely identifies plan. In contrast to AgentSpeak(L) the plan name in AgentSpeak(RT) is a required slot. It allows the agent to associate the corresponding execution time profile to the plan. We will explain the execution time profile of the plan in Section 5.4.3.

Triggering event. The triggering event defines the purpose of the plan. There are two types of changes in an agent's mental attitudes: changes in beliefs which refer to information about its state and states of other agents, and changes in the agent's goals. The *triggering event slot* can contain an *achieve* goal or a belief change event. An *achieve* goal event shows that the plan can be used to fulfill some condition. The belief change indicates that the plan can be used to react to the environment changes. The list of *triggering events* are presented in Table 5.2. A plan whose triggering event slot contains an *achieve* goal event is called *goal-invoked*, while a plan whose triggering event slot contains belief change event is called *belief-invoked*.

Context. The plan context specifies constraints which have to be satisfied for the plan to be applicable (a plan is applicable if its belief context is true).

Body. The body of a plan specifies a sequence of actions and (sub)goals to respond to the triggering event. Actions are the basic operations an agent can perform to change its environment in order to achieve its goals. Actions are denoted by *action symbols* and are written $a(t_1, \ldots, t_n)$ where a is an action symbol and $t_1, \ldots, t_n$ are the (ground) terms given as arguments to the action.

Plans may also contain achievement and test (sub)goals that need to be accomplished in order to handle a particular event successfully. Achievement subgoals allow an agent to choose a course of actions as part of a larger plan on the basis of its current beliefs[3]. An achievement subgoal $!g(t_1, \ldots, t_n)$ gives rise to an internal goal addition event $+!g(t_1, \ldots, t_n)$ which may in turn trigger subplans at the next execution cycle. Test goals are evaluated against the agent's belief base, possibly binding variables in the plan.

For example, the plan

```
+litter(L,S) : at(robot,S) & not parked(C,S) <-
    pickup(L); move(trashcan); deposit(L).
```

causes the agent to remove litter from the parking space the agent is in if there is no car parked in the space.

The BNF for AgentSpeak(RT) plans is given in Table 5.3, where **a** is an action symbol.

[3]Note that while plan patterns can be used to implement declarative goals [19], we consider only procedural goals at the moment.

plan-spec	::=	*plan time-profile*
time-profile	::=	"time-profile" ":" *number*"," *number*"," *number*
plan	::=	"@" *plan-name event* [":" *context*] "<-" *body* "."
plan-name	::=	**string**
event	::=	"+" ["!"] *literal* | "-" *literal*
context	::=	*true* | *literal* ("&" *literal*)*
body	::=	*true* | *step* (";" *step*)*
step	::=	*action* | *goal*
goal	::=	"!" *literal* | "?" *literal*
action	::=	[“.”]**a** [“(” *term-list* “)”]

Table 5.3: BNF for AgentSpeak(RT) plans

5.4.2 Primitive Actions

Primitive actions are the basic operations an agent can perform to change its environment in order to achieve its goals. Performing an action typically results in changes in the agent's beliefs when the action's effects on the environment are sensed at subsequent cycles of the interpreter. We distinguish user-defined primitive actions and internal actions.

User-defined Actions

User-defined primitive actions are implemented as Java methods. AgentSpeak(RT) supports two mechanisms for defining primitive actions: writing a class which implements the `ExternalAction` interface, and direct invocation of methods in existing legacy Java code. Each primitive action returns a boolean value. The `TRUE` value indicates successful completion of an action. The `FALSE` value indicates failure of the action. An example of primitive action implementation is shown in Section A.3.1.

Internal Actions

As in Jason, internal actions are distinguished from user-defined primitive actions by having a '.' character at the beginning of the action name. Internal actions form part of the AgentSpeak(RT) implementation, and realise basic operations which are key to the operation of the agent. For example, the `.send` action is used for inter-agent communication; `.print` action permits output to the standard output device; `.exec` allows the agent to execute an external program; `.gettime` returns the current time in ms etc. The full list of internal actions is presented in the Section A.3.2. The internal action

```
.print(''Performing next action'').
```

5.4.3 Execution Time Profile

In order to determine whether a plan can achieve a goal by a deadline with a given level of confidence, each action and plan has an associated *execution time profile* which specifies the probability that the action or plan will terminate successfully as a function of execution time. We assume that plans can be arbitrarily interleaved, and the estimated execution time of a plan is independent of any other plans the agent is currently executing. The expected execution time for an action or plan ϕ at confidence level α is given by $et(\phi, \alpha)$. Figure 5.2 shows an example of the execution time profile.

We assume that expected execution times increase monotonically with α, i.e., in general, to have higher confidence that a plan will complete successfully, we need to allow more time for the plan to execute. The shape of the execution time profile will typically be influenced by the (assumed) characteristics of the environment in which the agent will operate; for example, the probability of a plan to move to a location

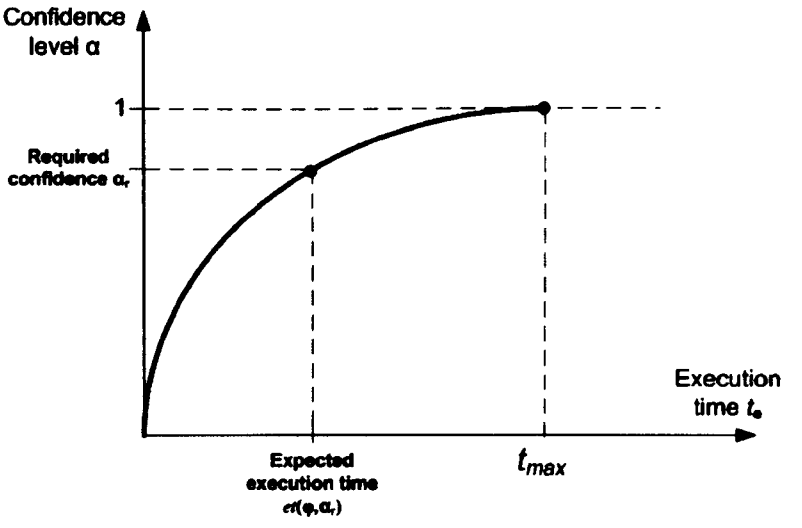

Figure 5.2: An execution time profile

terminating successfully within a given time may be lower in environments with many obstacles than in environments with fewer obstacles. Execution time profiles can be derived from an analysis of the agent's actions, plans and environment, or using automated techniques, e.g., stochastic simulation.

In the simple case of plans consisting of a sequence of actions, the execution time profile for the plan can be computed from the execution time profiles of its constituent actions. However, for plans that contain subgoals, the execution time profile will depend on the relative frequency with which the possible plans for a subgoal are selected in the agent's task environment.

The execution time profile t_e for an action or plan ϕ in AgentSpeak(RT) is specified as a power function with three parameters: k, ρ, α_{max}. Equation 5.1 represents a generic time profile. An agent developer can specify an execution time profile for a

plan or an action by varying the parameters.

$$\alpha(t_e) = f(\phi, t_e, k, \rho, \alpha_{max}) = \begin{cases} \left(\frac{t_e}{\rho}\right)^k & t_e \leq \rho \\ \alpha_{max} & t_e > \rho \end{cases}, \tag{5.1}$$

where t_e is expected execution time of an action or plan ϕ.

The parameter k is a positive number, which defines the inflection of the execution time profile curve. The parameter ρ is a positive number, which allows to scale the profile curve along the execution time axis. The parameter α_{max} is a maximal possible confidence level in a particular agent environment.

5.5 Intentions

The intention structure contains plans that have been chosen to achieve top-level goals or to respond to changes in the agent's beliefs. Plans triggered by changes in beliefs or the acquisition of an external (top-level) achievement goal give rise to new intentions. Plans triggered by the processing of an achievement subgoal in an already intended plan are pushed onto the intention containing the subgoal. Each intention consists of a stack of partially executed plans, a set of substitutions for plan variables, and a deadline and a priority. The set of variable substitutions for each plan in an intention results from matching the belief context of the plan and any test goals it contains against the agent's belief base. Also achievement goals can instantiate variables. The deadline and priority of an intention are determined by the triggering event of the root plan.

5.6 The Interpreter

The interpreter (Figure 5.3) is the main component of the agent. It manipulates the agent's belief base, event queue and intention structure, deliberates about which plan to select in response to goal and belief change events, and schedules and executes intentions.

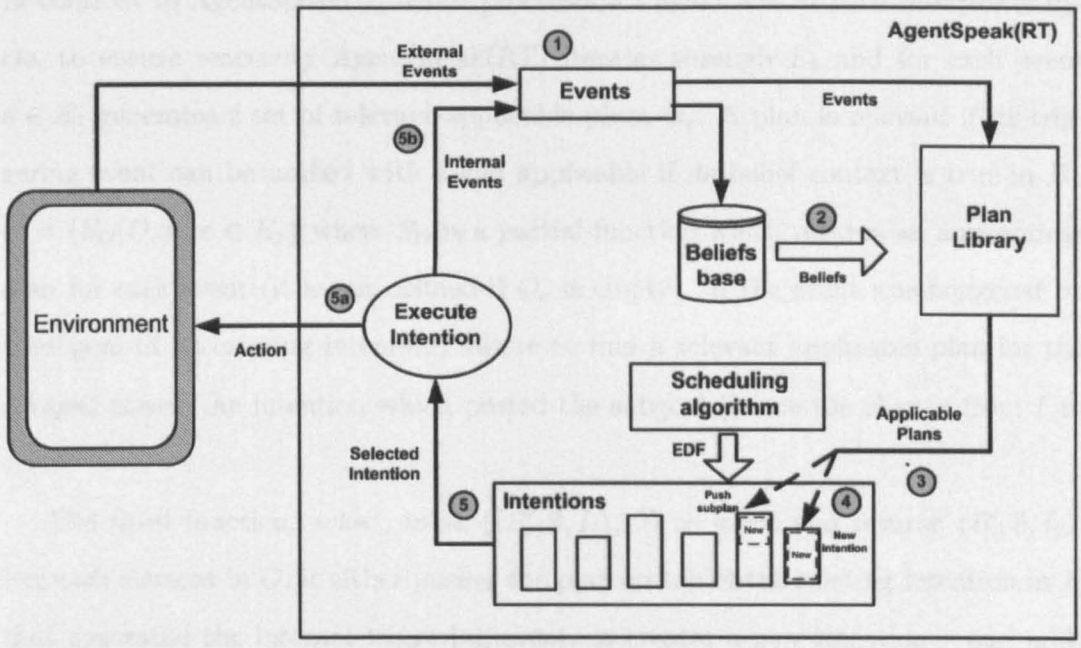

Figure 5.3: An AgentSpeak(RT) agent interpreter

The agent's state is a tuple $\langle B, E, I \rangle$ consisting of a set of base beliefs B, a set of events E, and an (ordered) set of intentions I. We can formalize the execution of the interpreter as a function which computes the new state of the agent and the executed action based on its current state and its inputs at the current cycle

$$(\langle B', E', I' \rangle, a) = exec(sched(opt(evt(\langle B, E, I \rangle, P, G))))$$

where P is a set of percepts, G is a set of goal addition events, B', E', I' are the updated belief, event and intention sets, and a is an action.

The function *evt* generates a set of events based on the agent's percepts and (external) goal addition events. It updates the belief base B with the percepts in P to give an updated belief base B' and a set of belief addition and removal events E_P, and returns a new state $\langle B', E_1 = E \cup E_P \cup G, I \rangle$.

The second function, *opt*, takes $\langle B', E_1, I \rangle$ as input and returns a pair $(\langle B', \emptyset, I_1 \rangle, O)$. In contrast to AgentSpeak(L) which processes a single event at each interpreter cycle, to ensure reactivity AgentSpeak(RT) iterates through E_1 and for each event $e \in E_1$ generates a set of relevant applicable plans O_e. A plan is relevant if its triggering event can be unified with e and applicable if its belief context is true in B'. $O = \{S_O(O_e) \mid e \in E_1\}$ where S_O is a partial function which returns an appropriate plan for each event (it is not defined if O_e is empty). If the event was triggered by a subgoal of an existing intention, failure to find a relevant applicable plan for the subgoal aborts the intention which posted the subgoal (hence the change from I to I_1).

The third function, *sched*, takes $(\langle B', \emptyset, I_1 \rangle, O)$ as input and returns $\langle B', \emptyset, I_2 \rangle$. For each element in O, it either pushes the plan on top of the existing intention in I_1 that generated the internal triggering event, or creates a new intention τ and adds it to a set I_E. I_2 is the result of applying the scheduling algorithm (see below) to $I_1 \cup I_E$. The scheduling algorithm returns a set of feasible intentions in deadline order (earliest first).

Finally, *exec* takes $\langle B', \emptyset, I_2 \rangle$ as input and returns a pair $(\langle B', E', I' \rangle, a)$, where I' is the result of executing the first intention in the schedule I_2, E' contains any internal goal addition event generated by executing the intention, and a is the action executed (or null if no action was executed). Executing an intention involves executing the first goal or action of the body of the topmost plan in the stack of partially executed plans which forms the intention. Executing an achievement goal adds a corresponding

internal goal addition event to E' and removes the achievement goal from the body of the plan. Executing a test goal involves finding a unifying substitution for the goal and the agent's base beliefs. If a substitution is found, the test goal is removed from the body of the plan and the substitution is applied to the rest of the body of plan. If no such substitution exists, the test goal is not removed and may be retried at the next cycle. Executing an action results in the invocation of the corresponding Java code. If the action completes within its expected execution time $et(a, \alpha)$, it is removed from the body of the plan. Actions which time out are not removed and may be retried at the next cycle[4]. The executed action is returned as a.

The scheduling algorithm is shown below (Algorithm 5.1). The set of candidate intentions is processed in descending order of priority. A candidate intention is added to the schedule if it can be inserted into the schedule in deadline order while meeting its own and all currently scheduled deadlines. A set of intentions $\tau_1, \ldots, \tau_n$ is feasible if there exists a schedule where each intention is executed before its deadline. To check whether a schedule exists for a set of intentions ordered earliest deadline first, it suffices to check that for every scheduled intention τ_i: $\Sigma_{j \leq i}\, et(\tau_j, \alpha) - ex(\tau_j) \leq d(\tau_i)$ where $ex(\tau_j)$ is the time τ_j has spent executing up to this point, and $d(\tau_i)$ is the deadline for τ_i. That is, the sum of expected remaining execution time of intentions scheduled earlier than τ_i including τ_i itself is less that the deadline of τ_i. A set of tasks is feasible iff they can be scheduled earliest deadline first [22]. Intentions which are not feasible in the context of the current schedule or which have exceeded their expected execution time are dropped[5].

[4]Allowing test goals and actions to be retried is not critical, but means that successful execution of intentions is less dependent on precise characterization of the execution time profile of actions.

[5]The real time guarantees we prove in Chapter 6 still hold in some circumstances if intentions that exceed their expected execution time are not dropped, but it complicates the presentation. The basic idea is that an intention τ which has exceeded its expected execution time has its priority

Algorithm 5.1 AgentSpeak(RT) Scheduling Algorithm

function SCHEDULE(I)

n := SIZEOF(I)

Γ: array of intentions[n] := $\{\emptyset \ldots \emptyset_n\}$

for all $\tau \in I$ in descending order of priority **do**

Γ': array of intentions[n] := $\{\emptyset \ldots \emptyset_n\}$

$j := 1$

$i := 1$

while $\Gamma[i] \neq$ null & $i \leq n$ **do**

if $d(\Gamma[i]) \leq d(\tau)$ **then**

$\Gamma'[i] := \Gamma[i]$

$j := j + 1$

else

$\Gamma'[i+1] := \Gamma[i]$

end if

$i := i + 1$

end while

$\Gamma'[j] := \tau$

if Γ' is feasible **then**

$\Gamma := \Gamma'$

end if

end for

return Γ

end function

reduced to 0. τ will only be scheduled if, after scheduling all higher priority intentions, there is sufficient slack in the schedule to execute at least one step in τ before its deadline. Given sufficient slack in the schedule, τ can therefore still complete successfully. It will however be dropped if it

The scheduler returns a set of intentions which is 'maximally feasible' (no more intentions can be added to the schedule if the scheduled intentions are to remain feasible at the specified confidence level) and moreover, intentions which are dropped are incompatible with some scheduled higher priority intention(s). Scheduling in AgentSpeak(RT) is pre-emptive in that the adoption of a new high-priority intention τ_i may prevent previously scheduled intentions with priority lower than i (including the currently executing intention) being added to the new schedule.

Note that if deadlines and priorities are not specified for external events (and hence $d = \infty, p = 0$ for all intentions), $et(\phi, \alpha) = \infty$ for all $\phi, 0 \leq \alpha \leq 1$, the behaviour of an AgentSpeak(RT) agent defaults to that of a non real-time BDI agent.

5.7 A Case Study: Nuclear Power Plant

In this section we use a simple case study to demonstrate the advantages of AgentSpeak(RT). A nuclear power plant produces electrical energy for consumers. Power plant operators are often overwhelmed by the need to process a large amount of incoming information in limited time. In order to help the operators an intelligent agent can be used. The real-time intelligent agent monitors the plant feed water systems, which supply hot water to a steam generator, and the reactor temperature. Furthermore, the agent is able to perform a diagnostics of all plant systems. One example of such a system is the Advanced Plant Analysis and Control System (APACS) framework [39, 40]. APACS was designed as a generic agent framework to help power plant operators notice and diagnose failures in continuous processes. In this case study we provide an AgentSpeak(RT) version of an intelligent control system for a nuclear power station.

exceeds its deadline.

system for a nuclear power station.

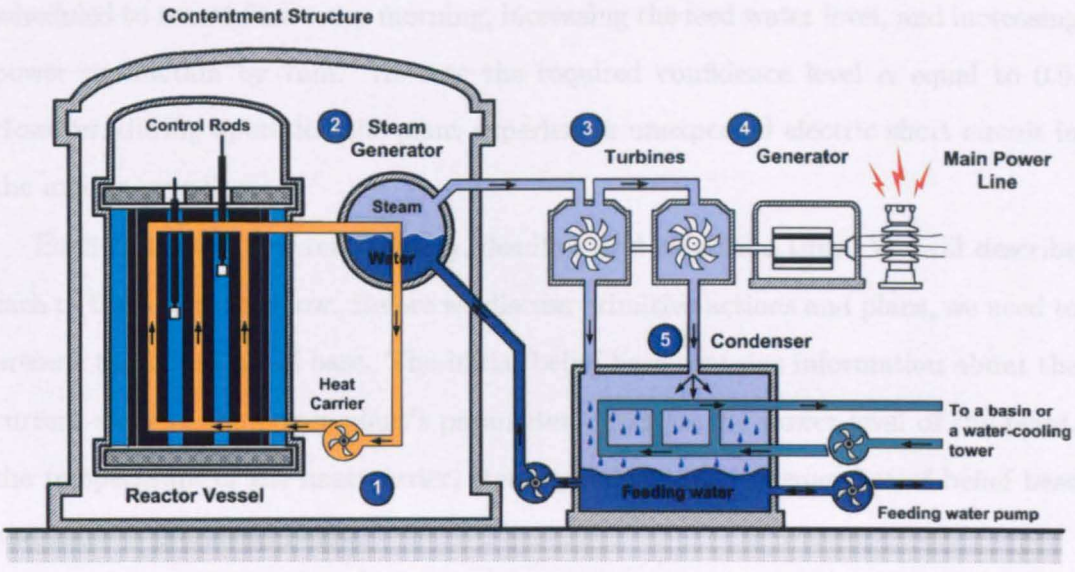

Figure 5.4: A Nuclear Power Plant scheme

The operation of a nuclear power plant with pressurized water reactors is shown in Figure 5.4. Energy released inside the *Reactor* propagates to a *heat carrier*. Then the heat carrier enters in the *Steam Generator*. The heat boils water and creates steam to turn a *Turbine*. As the *Turbine* spins, the *Generator* turns and its magnetic field produces electricity. At the end the steam enters a *Condenser*. It is cooled from water from a reservoir.

The agent provides the required electric power by monitoring the behaviour of the steam pressure and controlling the temperature of the carrier. The agent also handles unexpected situations, i.e. abrupt changes in workload, faults etc.

We intend to demonstrate how an AgentSpeak(RT) agent can be used to control a simplified model of the nuclear power station and show how to specify soft real-time requirements.

At the beginning of the working shift (at 6am), the nuclear plant operator gives

the agent three tasks: performing the power plant's systems diagnostics which is scheduled to run at 6am every morning, increasing the feed water level, and increasing power production by 7am. He sets the required confidence level α equal to 0.9. However, during operation the plant experiences unexpected electric short circuit in the main power line.

Each task has a different priority, deadline and execution time. We will describe each of them in turn below. Before we discuss primitive actions and plans, we need to present the initial belief base. The initial belief base contains information about the current state of the power plant's parameters, such as the power level of the plant, the temperature of the heat carrier, steam pressure, etc. A snapshot of belief base and goals is presented below.

```
Goals:
  !diagnostics          [6.50am, 4].
  !set-water-level(85)[6.55am, 10].
  !set-power-level(92)[6.56am, 10].


Beliefs:
  full-diagnostics.              //required type of diagnostics
  water-level(60).               //level of feed water[%]
  water-level-setpoint(85).      //setpoint of water level[%]
  steam-pressure(6).             //steam pressure[MPa]
  water-volume(50).              //volume of the water in the steam
                                 //generator[%]
  temperature(320).              //heating carrier temperature [C°]
  power-level(50).               //current power level of the plant [%]
```

```
status(power-line,ok)          //status of the main power line
status(reactor,ok).            //status of the reactor
status(turbines,ok).           //status of the turbines
status(steam-generator,ok).    //status of the steam-generator
status(elec-generator,ok).     //status of the electric-generator
status(feed-water-pump,ok).    //status of the feed water pump
status(cond-water-pump,ok).    //status of the condenser water pump


short-circuit[6.05am, 25].     //short circuit of main line event
```

There are several primitive actions available for the agent. These actions allow the agent to control the behaviour of the station.

check.

check(X) is a primitive action which allows the agent to check the status of the nuclear power station's subsystems e.g., the reactor, turbines, the steam generator, pumps etc.

change-water-level.

change-water-level(X) is a primitive action which changes the level of feed water using pumps. X is the desired level of feed water.

change-power-level.

change-power-level(X) is a primitive action which allows the agent to control the reactor's power. This action controls the nuclear reaction inside the reactor using control rods etc. X is the desired power level.

emergency-stop.

emergency-stop is a primitive action which runs emergency procedures to stop the reactor, the steam generator, turbines and pumps.

reset-plant.

`reset-plant` is a primitive action which resets all plant's systems after emergency stop.

Below, we give simple plans for dealing with the various tasks that the agent has to perform. The agent has two plans for diagnostics of the plant systems. The agent is able to perform a full diagnostics, which requires testing of all systems, and an express diagnostics which is used for testing only the main systems.

```
Plans:
  @full-diagnostics
  +!diagnostics : full-diagnostics <-
      check(reactor);
      check(power-line);
      check(turbines);
      check(steam-generator);
      check(elec-generator);
      check(feed-water-pump);
      check(cond-water-pump).

  @express-diagnostics
  +!diagnostics : express-diagnostics <-
      check(reactor);
      check(turbines);
      check(steam-generator);
      check(elec-generator).
```

These plans are triggered by a `+!diagnostics` goal event, and depending on the belief context ether the full diagnostics plan or the express diagnostics plan would

be chosen for that task. As we can see, in the current example, the AgentSpeak(RT) agent is asked to perform full diagnostics of the nuclear power station. The execution time of the plan, in that case, is equal to 45 minutes (see Figure 5.5).

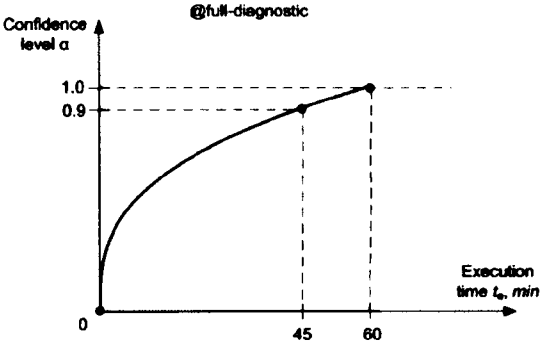

Figure 5.5: "Full diagnostics" plan's execution time profile (k=0.35, ρ=60, α_{max}=1)

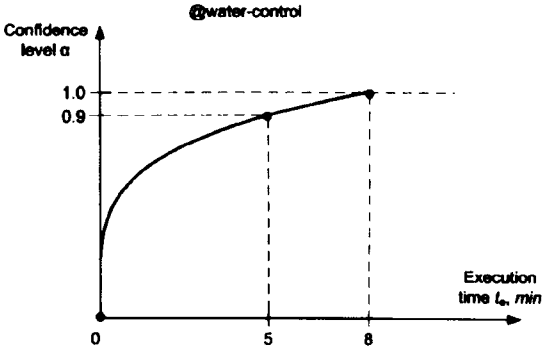

Figure 5.6: "Water control" plan's execution time profile (k=0.23, ρ=8, α_{max}=1)

Recall that the level of feed water is critical. In fact if the level drops below a certain threshold i.e., `water-level-setpoint`, the station would not be able to provide the required steam pressure. The agent has to change the level of the feedwater periodically.

```
@water-control
+!set-water-level(L) :  water-level(60)  <-
    change-water-level(L).

@norm-water-level
+!norm-water-level   : water-level-setpoint(L)  <-
    change-water-level(L).
```

In the current scenario, the agent has to manually change the water level from 60% to 90% of the full volume using plan `water-control`. Also the agent may normalise the level of feed water according a water level threshold i.e., `water-level-setpoint`. The execution time of the plan, in that case, is equal to 5 minutes (see Figure 5.6).

Plans `power-control` and `emergency-protection` below control the power produced by the nuclear power plant. It should be noted that the demand for power changes during the day; consequently, the power level of the nuclear station has to be changed accordingly. The plan `power-control` triggered by the event `+!set-power-level` executes the action `change-power-level` to change the reactor's power level. The execution time of the plan, in that case, is equal to 2 minutes (see Figure 5.7).

```
@power-control
+!set-power-level(K) : true <-
    change-power-level(K).

@emergency-protection
+short-circuit : true <-
    ?power-level(P)
    emergency-stop;
    !set-power-level(0);
    .print(''Main line power cut'');
    reset-plant;
    !set-power-level(P).
```

Moreover, the emergency protection plan executes emergency stop procedures to slow down the nuclear reaction and the steam generation, the power level is set on minimum and then restarts the plant. This plan also includes an internal action `.print` to inform the operator about the short circuit in the main line. The execution

time of the plan, in that case, is equal to 5 minutes (see Figure 5.8).

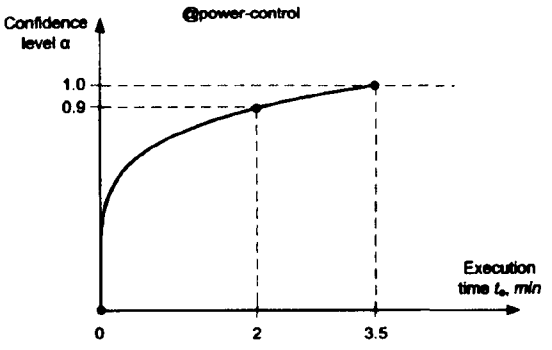

Figure 5.7: "Power control" plan's execution time profile (k=0.2, ρ=3.5, α_{max}=1)

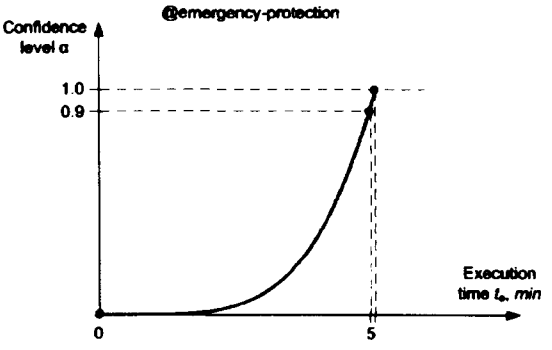

Figure 5.8: "Emergency protection" plan's execution time profile (k=4.5, ρ=5.1, α_{max}=1)

The agent's events are matched against the plan library. The plans `full-diagnostics`, `water-control` and `power-control` are executable and (individually) feasible and added to the set of candidate intentions. At 6am the unexpected short circuit of the main line event happens. This event triggers the plan `emergency-protection` to handle the emergency. As explained above (see Section 5.5), plans inherit their deadline and priority values from the top-level triggering event.

The `+short-circuit` event is extremely important at the current moment, and the agent has to be focused on it. Consequently, its intention has the highest priority (25), whereas the intentions to set *power level* and *feed water level* have medium priority (10). The intention to perform the *diagnostics* has the lowest priority (4). The scheduling algorithm (see Algorithm 5.1) then attempts to schedule the candidate intentions in decreasing order of importance.

The intention that corresponds to the `+short-circuit` event is therefore inserted into the schedule, which is initialised to empty. The intentions to increase water and

power level have same priority and are scheduled for execution in deadline order. However the agent is unable to complete the diagnostics by deadline because of the emergency, i.e., the intention `diagnostics` is infeasible. It can't be inserted in the schedule in deadline order together with other three intentions, and is dropped. It is important to note that the tasks will be executed one by one in earliest deadline order, which is why high priority intentions are executed first. Once the schedule has been computed, the interpreter executes one step of the first task, i.e., the `?power-level`(P) action, and starts a new cycle.

Assuming there are no belief or goal change events, at the next cycle the interpreter executes next step of the intention, i.e., the `emergency-stop` action and so on. In the same way, the AgentSpeak(RT) agent deals with the other tasks.

5.8 Summary

In this chapter we have presented AgentSpeak(RT), a real-time BDI agent programming language based on AgentSpeak(L). AgentSpeak(RT) extends AgentSpeak intentions with *deadlines* which specify the time by which the agent should respond to an event, and *priorities* which specify the relative importance of responding to a particular event. The AgentSpeak(RT) architecture provides a flexible framework for the development of real-time BDI agents. An AgentSpeak(RT) agent will pursue a priority-maximal set of intentions which can be achieved by their deadlines with a specified confidence level. If not all intentions can be achieved by their deadlines, the agent prefers intentions with greater priority.

By varying the level of confidence, the agent developer can control the degree of 'optimism' the agent adopts when determining the time required to complete a task in a given environment. Higher levels of confidence will typically result in the agent

allowing more time to complete a task, and cause fewer tasks to be scheduled in a given period of time. As tasks are scheduled in priority order, increasing the level of confidence required also has the effect of causing the agent to focus more on high priority tasks at the expense of lower priority tasks which might be achievable given a more optimistic view of execution time.

If no deadlines or priorities are specified, the behaviour of the agent defaults to that of a non real-time BDI agent. Real-time tasks can be freely mixed with tasks for which no deadline and/or priority has been specified by the developer or user. Tasks without deadlines will be processed after any task with a specified deadline, and for tasks with the same deadline, the agent will prefer tasks of higher priority. An example AgentSpeak(RT) agent program for the intelligent control of a nuclear power plant was also described.

In the next chapter we will discuss the real-time properties of an AgentSpeak(RT) agent and prove them.

Chapter 6

AgentSpeak(RT) Properties

In this chapter we show that an AgentSpeak(RT) agent is a real-time BDI agent in the sense defined in Chapter 2. Specifically we prove a number of properties of AgentSpeak(RT), including that the reactivity delay of an AgentSpeak(RT) agent is bounded, that it commits to a priority-maximal set of intentions, and that in a static environment its intentions will complete successfully by their deadlines with specified confidence. We also develop a model of the 'difficulty' of the agent's environment, and show how it can be used to determine the priority of intentions which will execute successfully by their deadlines with specified confidence.

6.1 Proof of Real-time Guarantees

In this section we show that under certain assumptions (which we believe are reasonable for real-time applications), the time required to execute a single cycle of the AgentSpeak(RT) interpreter (and hence the reactivity of the agent) is bounded. We also show that an AgentSpeak(RT) agent commits to a priority-maximal set of intentions, and that, given a fixed schedule, the probability that an intention will

complete successfully by its deadline is α.

We make the following assumptions about the agent's program and task environment:

1. the set of possible beliefs has a fixed maximal size (for example, the set of possible beliefs can be restricted to the set of ground instances of any atomic formula appearing in a belief context or a test goal for a finite set of constants);

2. the set of possible goals has a fixed maximal size (for example, the set of possible goals can be limited to the set of ground instances of any atomic formula appearing in an achievement goal for a finite set of constants);

3. the maximal possible interval between the arrival time and deadline of any event (with deadline $< \infty$) is a constant d_{max};

4. the minimal expected execution time for any plan is a constant t_{min}; and

5. there is a maximal expected execution time, t_{max}, for any action in the agent program (i.e., $t_{max} = max(et(a, \alpha))$ for any action a at the specified α).

Theorem 1. *If the sets of possible beliefs and goals, the maximal expected action execution time and the maximal distance to deadline have a fixed maximal size, and the minimal plan execution time has a fixed minimal size, then the time required to execute a single cycle of the AgentSpeak(RT) interpreter is bounded by a constant δ_c (sum of time bounds for the interpreter cycle steps).*

Proof. We have formalised the execution cycle in Section 5.6. The time required to compute *evt* depends on the size of the sets P and G. If the set of all possible beliefs is limited to a fixed finite set of ground belief atoms (assumption 1 above), then the number of possible percepts $|P|$ is bounded by a constant (assuming that the agent's percepts are limited to changes in its beliefs).

If the set of all possible agent goals is similarly limited (assumption 2), then the number of possible goals $|G|$ is also bounded by a constant. This means that $|E|$ and $|B|$ are also bounded by a constant at all stages of the cycle.

The time required to compute *opt* is bounded if $|B|$ is bounded. Computing the set of applicable plans for each event involves evaluating the belief context of each plan whose trigger matches the event against the agent's beliefs. Assuming that returning the set of plans which match an event is a constant time operation and matching the belief context of a plan against the agent's beliefs is bounded by a polynomial in $|B|$, if $|B|$ is bounded, then the time required to compute *opt* is also bounded by a constant.

The complexity of *sched* is $O(|I|^2)$. In the worst case, when priority varies with deadline and intentions are inserted into the schedule in order of decreasing deadlines, then the feasibility of each new intention involves checking the feasibility of all currently scheduled intentions. $|I|$ is bounded if the maximal possible interval between the arrival time and deadline of any event is a constant d_{max} (assumption 3), and the minimal expected execution time for any plan is a constant t_{min} (assumption 4). Then the maximal possible number of schedulable intentions is bounded by d_{max}/t_{min}.

By assumption 5, the maximum action expected execution time for an action and hence the time to compute *exec* is bounded by a constant t_{max}. □

By the *reactivity delay* of an agent we mean the time the system takes to recognize and respond to an external event [14] (i.e., the time from the arrival of the event to the selection of a plan for the event).

Theorem 2. *Given the assumptions 1 – 5 above and Theorem 1, the reactivity delay of an AgentSpeak(RT) agent is bounded.*

Proof. The maximum reactivity delay is for an event which arrives just after the evaluation of *evt* begins, which is guaranteed to be responded to by the end of the *next* agent cycle. Since the agent's cycle is bounded by δ_c, the maximum reactivity delay is hence bounded by $2\delta_c$. □

We now show that an AgentSpeak(RT) agent commits to a priority-maximal set of intentions.

Definition 1. *Consider a set of intentions I. A set $\Gamma \subseteq I$ is a priority-maximal set of intentions (with respect to I) if:*

1. *Γ is feasible;*
2. *$\forall \tau \in I$ such that $\tau \notin \Gamma$: $\{\tau\} \cup \Gamma$ is infeasible;*
3. *$\forall \tau \in I$ such that $\tau \notin \Gamma$, either $\{\tau\}$ is infeasible, or $\exists \Gamma' \subseteq \Gamma$: the minimal priority of an intention in Γ' is greater or equal to the priority of τ $p(\tau)$ and $\Gamma' \cup \{\tau\}$ is infeasible.*

Intuitively, this definition describes a subset of I which is 'maximally feasible' (no more intentions from I can be added if the intentions are to remain feasible at the specified confidence level) and moreover, intentions in $I \setminus \Gamma$ are incompatible with some subset of Γ which contains intention(s) of the same or higher priority. Observe that if all intentions in I have a unique priority, then there is only one priority-maximal subset of Γ, containing the maximal number of highest priority intentions which are jointly feasible[1].

[1] In general, a priority-maximal set of intentions is not guaranteed to contain the largest number of high priority intentions. For example, if $S = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, where $p(\tau_1) = p(\tau_2) = p(\tau_3) = 2$, $p(\tau_4) = 1$, and it is possible to schedule either τ_1 and τ_4 together, or τ_2 and τ_3 together, both sets $\{\tau_1, \tau_4\}$ and $\{\tau_2, \tau_3\}$ will be priority-maximal sets (but, for example,$\{\tau_1\}$ will not be). Computing

Theorem 3. *Given a partially ordered set of intentions* $I = \{\tau_1, \tau_2, \ldots, \tau_n\}$, *where* $p(\tau_i) \geq p(\tau_j)$ *for* $i < j$, *the AgentSpeak(RT) scheduling algorithm generates a priority-maximal set of intentions* $\Gamma \subseteq I$.

Proof. Note that the AgentSpeak(RT) scheduling algorithm generates a sequence of sets starting with $\Gamma_0 = \emptyset$, and sets Γ_i to be $\Gamma_{i-1} \cup \{\tau_i\}$, $\tau_i \in I$ if $\Gamma_{i-1} \cup \{\tau_i\}$ is feasible in deadline order, or Γ_{i-1} otherwise. The last set Γ_n is Γ.

By construction, Γ is a feasible set of intentions. Γ is also clearly a maximally feasible subset of I: there is no $\tau \in I$ such that $\tau \notin \Gamma$ and $\Gamma \cup \{\tau\}$ is feasible. To prove that it is priority-maximal, let $\tau_i \in I$, $\{\tau_i\}$ feasible, and $\tau_i \notin \Gamma$. We need to show that τ_i is incompatible with some subset of Γ which contains only intentions of the same or higher priority than $p(\tau_i)$. Since the intentions are added to Γ in descending order of priority, when τ_i is considered and found incompatible with Γ_{i-1}, $p(\tau_i) \leq \min(\{p(\tau') : \tau' \in \Gamma_{i-1}\})$. □

Theorem 4. *The probability that an intention* τ *will execute successfully in a static environment is equal to* α.

Proof. Immediate, from the fact that the execution time profiles of plans give us the estimate of duration of the task with the probability α. □

6.2 Dynamic Environments

The guarantees in the previous section are for a static environment – they consider only the probability that a scheduled intention completes successfully or aborts. They do not consider cases where a scheduled intention is dropped as a result of the arrival of a higher priority task.

the set containing the largest number of highest priority intentions is a hard combinatorial problem, which can not be solved by a real-time scheduler.

In this section, we develop a simple model of task arrival which can be used to characterise the 'difficulty' of an agent's task environment. Later we show how this model can be used to determine the priority of intentions which can be reliably scheduled, and to estimate the probability that a scheduled intention of a given priority will be displaced from the schedule by the arrival of an intention of higher priority and the probability the task will be executed by its deadline.

We characterise the task environment in terms of the average arrival rate and time available for the execution of intentions of a given priority. Let r_i be the average triggering rate of intentions of priority i (expressed as the number of triggering events / unit time), and a_i the average time available for their execution, i.e., the difference between the intention's deadline and the time at which it was triggered. For example, if each external achievement goal has a distinct priority level, r_i and a_i correspond to the arrival rate and average time to achieve that particular type of goal. We assume that $a_i \geq t_i$ where t_i is the average execution time of intentions of priority i at the specified confidence level, i.e., that deadlines advance the time at which intentions are triggered such that intentions are always individually feasible on average. t_i can be computed from the execution time profiles of the plans in the agent's plan library.

Clearly, the larger r_i and the smaller the difference between a_i and t_i, the more difficult the agent's environment. The larger the value of r_i, the lager the number of intentions the agent must execute in a given period of time; the smaller the value of $a_i - t_i$, the less time there is to accommodate intentions of priority less than i. In general, the probability that an intention of priority j will be unschedulable is an increasing function of r_i and decreases with $a_i - t_i$ for all $i > j$.

According to Little's law [23], in the worst case, when the schedule is full and intentions complete their execution just before their deadlines, the long term average

number of intentions of priority i in the agent's schedule is given by

$$\lambda_i = r_i \times a_i \tag{6.1}$$

The amount of uncommitted or 'slack' time unused by intentions of priority i in such a schedule is $s_i = \lambda_i \times (a_i - t_i)$. We assume that $s_i \geq 0$ for all i given an otherwise empty schedule, i.e., that the average arrival rate and time available for execution of intentions of each priority level are feasible for the agent. For intentions of priority $i-1$ to be reliably scheduled, the total time required for their execution, $\lambda_{i-1} \times t_{i-1}$, must be less than s_i. If the maximum priority of any intention is m, then the time available to schedule intentions of priority j is

$$s_{j+1} = \lambda_m \times a_m - \sum_{k=j+1}^{m} \lambda_k \times t_k \tag{6.2}$$

Hence intentions of priority $j < m$ are typically unschedulable if $s_{j+1} \ll \lambda_j \times t_j$. For given values of r_i, a_i and t_i, we can therefore determine the priorities of intentions which can be typically scheduled.

6.2.1 Probability of Scheduling an Intention

We can estimate the probability that the intention τ of priority j is scheduled, $F(\tau_j)$, using the dynamic model above. The slack, which can be assigned for intentions of particular priority j is given by Equation 6.2.

For a new intention of priority j to be schedulable, there must be at least t_j slack in the schedule at priority level j, i.e., $s_j \geq t_j$. The amount of slack at priority level j in the schedule depends on the number of intentions in the schedule at priority levels $j, \ldots, m$. (A new intention of priority j can displace already scheduled intentions with priority $< j$ but not already scheduled intentions of priority j or higher.) Any currently scheduled intentions of priority i, $j \leq i \leq m$, must have arrived in the last a_i

time units, i.e., between $-a_i$ and now. The number of intentions of each priority level $j, \ldots, m$ arriving between times $-a_j, \ldots, -a_m$ and now can be represented as a vector $\langle f_j, \ldots, f_m \rangle$ where f_i for $i \in \{j, \ldots, m\}$ is the number of intentions of priority i which arrived within the last $-a_i$. Thus, for an intention of priority j to be schedulable, the following must hold:

$$t_j \leq f_m \times (a_m - t_m) - \sum_{i=j}^{m-1} f_i \times t_i \tag{6.3}$$

Let the set of vectors satisfying this condition be

$$F = \{\langle f_j, \ldots, f_m \rangle : \; f_m \times (a_m - t_m) - \sum_{i=j}^{m-1} f_i \times t_i \geq t_j\}$$

The probability that an intention of priority j is schedulable, F_j, is then the probability that at most the number of intentions of each priority level specified by one such sequence of arrivals occurs. If the arrival of triggering events is a Poisson process[2], the probability that at most f_i intentions are added to the schedule in time a_i is given by

$$F(f_i) = \sum_{x=0}^{f_i} \frac{e^{-\lambda_i} \lambda_i^x}{x!}$$

That is, the probability that exactly 0 or 1 or 2 or ... or f_i intentions are added to the schedule in an interval of length a_i. The probability that at most the number of intentions of each priority level specified by one such sequence occurs is then

$$F(\tau_j) = 1 - \prod_{\langle f_j, \ldots, f_m \rangle \in F} (1 - F(f_j) \times \ldots \times F(f_m)) \tag{6.4}$$

[2]Poisson process is one of the most important models used in queueing theory. It allows to determine the probability of a number of events occurring in a fixed period of time if these events occur with a known average rate and independently of the time since the last event.

6.2.2 Probability of an Intention Displacement

We can also determine the probability that a scheduled intention of priority j (assumed to be reliably schedulable) is displaced from the schedule by the arrival of a higher priority intention. If the uncommitted time at priority m, s_m, is sufficient to schedule the expected number of intentions of priority $m-1$, then for an intention of priority $m-1$ to be displaced from the schedule, $u_m = \lceil s_{m-1}/t_m \rceil$ intentions must be added to the schedule during time a_{m-1}. The expected number of priority m intentions arriving in time a_{m-1} is $\lambda_m = r_m a_{m-1}$. If the arrival of intentions is a Poisson process, the probability that at least u_m intentions are added to the schedule in time a_{m-1} is given by

$$U(u_m) = 1 - F(u_{m-1}).$$

That is, 1− the probability that exactly 0 or 1 or 2 or ... or u_{m-1} events arrive in an interval of length a_{m-1}.

In general, for a scheduled intention of priority $j < m$ to be displaced, sufficient intentions of priority $> j$, with total execution time $> s_j$, must arrive within a time interval a_j.

A set of intentions with priorities $j+1, j+2 \ldots, m$ sufficient to displace an intention of priority j can be represented as a vector $\langle u_{j+1}, \ldots, u_m \rangle$ where $u_i \in \{j+1, \ldots, m\}$ is the number of intentions of priority i that arrive within a_j. To displace an intention of priority j such vectors must satisfy a number of conditions. First, the number of intentions of each priority must be feasible given s_j. Second, the combined execution time of all intentions in the set must be greater than s_j. Third, that the combined execution time of the intentions should exceed s_j by at most the least execution time of any intention in the set. That is, all possible sequences of intentions of priority $> j$ which have combined execution time "just greater" than s_j. Let the set of vectors

satisfying the conditions be

$$\begin{aligned} U \quad = \{\langle u_{j+1}, \ldots, u_m \rangle : \quad & 0 \leq u_i \leq s_j/t_i, \\ & \textstyle\sum u_i t_i > s_j, \\ & \textstyle\sum u_i t_i - \min_{i \in \{j+1,\ldots,m\}}(t_i) \leq s_j\} \end{aligned}$$

The probability that an intention of priority j is displaced, $U(\tau_j)$, is then the probability that at least the number of intentions of each priority level specified by one such sequence occurs:

$$U(\tau_j) = 1 - \prod_{\langle u_{j+1},\ldots,u_m \rangle \in U} (1 - U(u_{j+1}) \times \ldots \times U(u_m)) \tag{6.5}$$

where $U(u_i) = 1 - \sum_{x=0}^{u_i - 1} \frac{e^{-\lambda_i} \lambda_i^x}{x!}$ as above.

For different applications and priority levels, different probabilities of displacement may be appropriate. If, for the intended application, U_j is deemed to be too high, the agent developer must either reduce the average triggering rate of intentions of priority $> j$ or increase $a_i - t_i$ for $i \geq j$, e.g., by reducing the execution time of the agent's plans.

6.2.3 Probability of the Successful Execution of an Intention

In this section we show how to compute the probability that a task that has been given to the agent will be completed by its deadline. The probability that an intention τ of priority j will be executed by its deadline, depends on 3 probabilities: 1) the probability that the intention will be reliably scheduled, 2) the probability that the intention is not displaced by intentions of priority $> j$; and 3) the probability that the intention will execute successfully in a static environment.

We have already derived the probability that the intention τ of priority j will be displaced $U(\tau_j)$ in the previous section (see Equation 6.5). Therefore the probability

that intention τ_j will not be displaced is equal to $1 - U(\tau_j)$. The probability that intention τ_j will be scheduled is given by Equation 6.4. Also recall that the probability that the intention τ_j should complete successfully before its deadline is, in fact, the confidence level α.

The probability that an AgentSpeak(RT) agent will execute an intention τ of priority j to completion is the probability that the intention will be scheduled, will not be displaced by higher priority intentions, and will execute successfully, as follows:

$$E(\tau_j) = F(\tau_j) \times (1 - U(\tau_j)) \times \alpha. \tag{6.6}$$

For given values of r_j, a_j and t_j, we can determine the probability that an intention of priority j related to a goal achievement or a belief change event will be executed by its deadline. In the nuclear power plant example, for a given rate of changes to the volume of feed water and the time it takes to execute the agent's plans to change the water level, we can determine the probability that a goal to change the level of feed water will be executed by the deadline.

6.3 Summary

In this chapter we have stated and proved real-time properties of an AgentSpeak(RT) agent, such as guaranteed reaction time of the AgentSpeak(RT) interpreter and probabilistic guarantees of successful execution of intentions. We have also developed a simple model of the 'dynamism' of the agent's environment, and shown how it can be used to determine the priority of intentions that can be reliably scheduled, and the probability that a scheduled intention of given priority will be completed by its deadline. In the next chapter we intend to show how to extend the AgentSpeak(RT) architecture to allow parallel execution of intentions.

Chapter 7

AgentSpeak(RT) with Parallel Execution of Intentions

A major advantage of BDI-based agents is their ability to pursue multiple intentions in parallel. While it offers real-time guarantees, the AgentSpeak(RT) architecture described in Chapter 5 executes intentions sequentially, i.e., one intention at a time. In this chapter we present a multitasking approach to the parallel execution of intentions in the AgentSpeak(RT) architecture. The parallel execution of intentions allows the agent to pursue multiple goals at the same time. We also prove real-time properties of AgentSpeak^{MT}(RT), a multitasking version of AgentSpeak(RT) architecture, and demonstrate the advantages of parallel execution of intentions in AgentSpeak(RT) by showing how it improves the behaviour of the example intelligent control system for a nuclear power station from Chapter 5.

7.1 Shared Resources

The problem with parallel execution of intentions is that there can be undesirable interactions between plans. Possible interactions between plans are defined in terms of shared resources. We define a resource as a physical or virtual component of limited availability within an agent. Each agent has a finite set of shared resources $res_1, res_2, \ldots, res_n$ which are used during its operation. A shared resource may be physical, e.g., picking up litter may require the use of a gripper, or logical, e.g., a mutual exclusion requirement that incompatible actions are not executed at the same time.

Executing an action requires exclusive access to zero or more of the shared resources $res_1, res_2, \ldots, res_n$, and the set of resources required to execute a plan is the union of the resources required to execute each of its actions. The set of resources required to execute an action (and hence a plan) forms part of the specification of the action for a particular hardware or software agent platform.

Shared resources can be used by only one plan at a time, and are assumed to be reusable: after the plan has been executed the resource becomes available for the execution of other plans. If two plans require the same shared resource, the execution of the intentions containing the plans must be serialised, as explained below. Only intentions that do not have resource conflicts may execute concurrently. In AgentSpeak^{MT}(RT) a developer can specify a required level of confidence for the successful execution of intentions in terms of two probabilities, α and β. An AgentSpeak^{MT}(RT) agent should schedule an intention so as to ensure that the probability that it does not compete for resources with other agent's intentions is at least β. If not all intentions can be executed with the required level of confidence due to lack of time or resource conflicts, the agent favours intentions responding to high

priority events.

plan-spec	::=	*plan time-profile resource-profile*
time-profile	::=	"time-profile" ":" *number*"," *number*"," *number*
resource-profile	::=	"resource-profile" ":" (resource"," *number*";")*
plan	::=	"@" *plan-name event* [":" *context*] ["::" *resources*]
		"<-" *body* "."
plan-name	::=	**string**
event	::=	"+" ["!"] *literal* | "-" *literal*
context	::=	*true* | *literal* ("&" *literal*)*
resources	::=	*resource* ("," *resource*)*
resource	::=	**r** ["(" *term-list* ")"]
body	::=	*true* | *step* (";" *step*)*
step	::=	*action* | *goal*
action	::=	["."]**a** ["(" *term-list* ")"]
term-list	::=	*term* ("," *term*)*
term	::=	*constant* | *variable* | *function*

Table 7.1: BNF for AgentSpeak^{MT}(RT) plans

Recall that each action and plan has an associated execution time profile which specifies the probability that the action or plan will terminate successfully as a function of execution time (See Chapter 5). In order to determine if plans can execute concurrently, each plan is also associated with an *execution resource profile* which specifies the probability that the plan will require each of the agent's shared resources $res_1, res_2, \ldots, res_n$. The BNF for AgentSpeak^{MT}(RT) plans is given in Table 7.1.

7.2 Plan-Resource Tree

Execution resource profiles can be derived from an analysis of the agent's actions, plans and environment, or using automated techniques, e.g., stochastic simulation. One way to analyse the relation between agent's resources, actions and plans, and to determine the probability that a particular resource will be required by the execution of an intention is to use a 'Plan-Resource Tree'. A *plan-resource tree* (PRT) is a modification of a *goal-plan tree* (GPT) [35].

A GPT is a bipartite directed graph, connecting goals or subgoals with plans and plans with subgoals. Thangarajah *et al.* introduced goal-plan trees in [35] as a way to represent the interaction between plans and goals in BDI agent programming languages. For a plan in the tree to be accomplished, all of its subgoals must be achieved. However, to achieve a (sub)goal only one of its relevant plans needs to be accomplished. Each tree node (i.e., a goal or plan) is associated with summary information about required resources. Thangarajah et al. consider both reusable and consumable resources. Reusable resources can only be used by one plan at a time, and after the plan has been executed, the resources become available for the execution of other plans, e.g., CPU, communication channel. In contrast, consumable resources are those which can only be used once and are consumed by the execution of a plan, e.g., time, energy.

An example of a goal-plan tree (without resource summary) is shown in Figure 7.1. The agent has two top-level goals G_1 and G_2. There are three alternative plans Pl_1, Pl_2 and Pl_3 for achieving goal G_1. Each plan has different resource requirements R_1, R_2 and R_3. Plans Pl_1 and Pl_3 have subgoals SG_{11} and SG_{12} and so on. In order to achieve the top-level goals G_1 and G_2 only one of their alternative plans has to be accomplished.

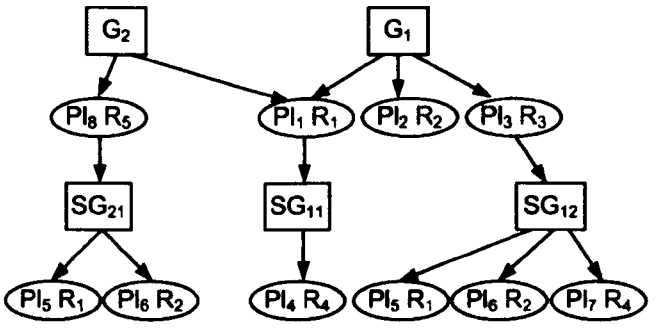

Figure 7.1: A goal-plan tree structure

Resources are not attached to each individual action, rather a resource summary is specified for a plan, which includes resource requirements of the subgoals and actions in the plan body. The resource requirements for a goal are combined resource requirements of all relevant plans for this goal. The summary resource requirements for each top-level goal are generated using the GPT, summing up the requirements starting at the leaves. The resource summary consists of a set of *necessary* resources (i.e., minimum resource requirements) and a set of *possible* resources (i.e., maximum resource requirements).

The resulting resource summary is used to determine resource conflicts, so they can be avoided by choosing alternative plans or appropriately scheduling plan execution. An algorithm is provided to determine whether an agent can adopt a new goal with respect to its existing set of goals. The initial GPT and summary information for each goal type are generated at compile time, and identified resource conflicts are monitored at runtime in order to avoid them.

Negative interactions between goals such as competition for resources can in result failure of the goals. On the other hand positive interactions between goals allow the agent to take advantage of situations where goals may have common subgoals. In [34, 33], Thangarajah *et al.* explored mechanisms for identifying potential common

subgoals and interference between the goals. They also presented mechanisms for scheduling to take advantage of the positive interactions and to avoid interference between goals. The positive and negative interaction between goals are identified by generating definite and potential effects of (sub)goals and plans to achieve the goals. The effects of executing a plan are the effects of executing the actions within the plan represented as logical conditions. As before effects summaries are derived by propagating the effects of plans and (sub)goals up the goal-plan tree. These summaries are used to reason about merging plans and avoiding interference between them.

In [32] Thangarajah *et al.* reported the evaluation of the costs and the benefits of reasoning about resource requirements as presented in [35]. The results show an increase in the number of goals successfully achieved and a small increase in computational costs. However as a goal-plan tree grows, the amount of summary information could potentially grow exponentially, which in case of large problems would significantly increase computational costs and afect the overall performance of the agent.

Shaw and Bordini in [30] mapped GPTs to Petri nets to avoid the need for summary information. The Petri-net based technique solved the problem of summary information growth and can be used to reason about both positive and negative interaction between goals. In the Petri net, goals and plans are represented by a series of places and transitions. Goals are linked to the relevant plans. Plans and subgoals are nested within each other, similarly to the GPT.

In [31], Shaw and Bordini provided mechanisms for reasoning about agent resources being combined with reasoning for positive and negative interactions between the goals combining various types of GPT techniques. Whilst the Petri net approach allows to avoid the use of summary information when reasoning about positive and

negative interaction between goals, this approach does not work when reasoning about resources. For reasoning about resources a compact form of summary information is used. The summary information of a GPT is used in two ways: 1) to store the summary of all resource requirements and to decide if a goal can be adopted based on existing resource availability; and, 2) where a goal or subgoal has several applicable plans, to provide summary information just for the subtrees. The summary information at the root of the tree gives the minimum and maximum resource requirements for each goal. The Petri-net is used to keep a summary information for current goals, and checks if there are sufficient resources available to adopt a new goal. The experimental results show a significant increase in the number of goals achieved with little additional reasoning cost.

While the proposed GPT techniques allow an agent to resolve resource conflicts between goals, the summary information requires exponential time to generate. Moreover, the view of resources used in these techniques is different from that considered in this section, in that the probability that a plan would require a shared resource is not considered.

Our approach determines an execution resource profile (i.e., probabilistic resource requirements) for each plan using a GPT-like structure, and does not require to update the summary information in runtime. For each resource we need to determine the probability that it will be required by an intention: we can compute this by turning the goal-plan tree into plan-resource tree (PRT) structure (see Figure 7.2) which represents the relationship between plans and the resources they require.

A PRT is also a bipartite directed graph, but in contrast to a GPT, it connects plans with resources required by the corresponding plans and resources with (sub)plans. PRTs also represent the probability that a sub-plan will be chosen for a sub-goal within a plan. Obviously, the probability that a plan will require each of the

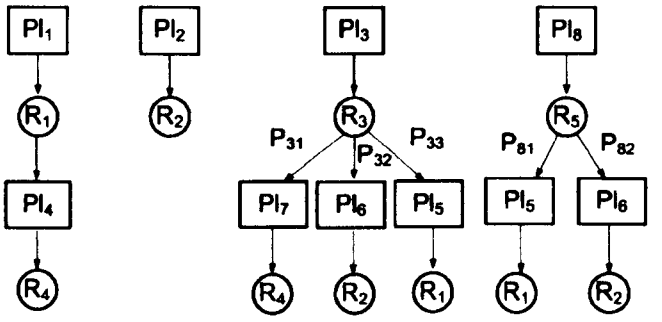

Figure 7.2: Plan-Resource Trees

shared resources required by each of the actions in the plan is 1. However for plans which contain subgoals, the execution resource profiles will depend on the relative frequency with which the alternative plans for a subgoal are selected in the agent's task environment.

We can determine a execution resource profile for each plan using the corresponding plan-resource tree structure (PRT). The tree is traversed from root to leaves, calculating the resource requirements and corresponding probabilities. For a plan Pl_i with the corresponding PRT if a shared resource R_u is used by possible (sub)plans Pl_j to Pl_k of Pl_i then the probability that R_u is required by Pl_i is equal to $1 - \prod_{n=j..k} (1 - P_{i,n})$, where $P_{i,n}$ is the probability that the (sub)plan Pl_n will be selected. For example, plan Pl_3 in Figure 7.2 is associated with the execution resource profile: $\langle R_1, P_{33};\ R_2, P_{32};\ R_3, 1;\ R_4, P_{31};\ R_5, 0; \rangle$. An agent's PRT can be computed offline and only needs to be computed once.

7.3 Multitasking Reasoning

In contrast to AgentSpeak(RT), the AgentSpeak^{MT}(RT) scheduling algorithm allows an agent to execute several intentions apparently simultaneously. In the case

of an agent with a single CPU, it can only execute one step of any intention at a time, which means that the agent must switch from one intention to another. Hence, if the agent is able to switch between the tasks frequently enough, i.e., if the duration of the single agent's execution cycle is small, then the impression of parallelism is achieved.

The AgentSpeak^{MT}(RT) interpreter is similar to the interpreter of AgentSpeak(RT) (see Section 5.6), however the function *sched* and the function *exec* are different.

7.3.1 The AgentSpeak^{MT}(RT) Scheduler

An AgentSpeak^{MT}(RT) schedule is a set of partially overlapping intentions τ_1, ..., τ_n. The scheduling algorithm has to distinguish between intentions that can be scheduled sequentially and intentions that can be scheduled concurrently.

A set of intentions is a set of pairs $\{(s_1, \tau_1), \ldots, (s_n, \tau_n)\}$ where s_i is the time at which intention τ_i will next execute. A schedule is feasible if each intention will complete execution before its deadline with a probability α and with a probability of resource conflict less than $1-\beta$. More precisely, a set of intentions $\{(s_1, \tau_1), \ldots, (s_n, \tau_n)\}$ is feasible if

1. for each scheduled intention (s_i, τ_i):

$$s_i + et(\tau_i, \alpha) - ex(\tau_i) \leq d(\tau_i), \tag{7.1}$$

   where $ex(\tau_i)$ is the time τ_i has spent executing up to this point, and $d(\tau_i)$ is the deadline for τ_i;

2. the probability that there is no resource conflict with intentions $\{(s_j, \tau_j) \mid s_i < s_j + et(\tau_j, \alpha) \wedge s_j < s_i + et(\tau_i, \alpha)\}$ which are already scheduled and execute

concurrently with τ_i is at least β:

$$\prod_{\tau_j} \prod_{u=1}^{n} (1 - P(\tau_i, res_u) \times P(\tau_j, res_u)) \geq \beta, \tag{7.2}$$

where $P(\tau_i, res_u)$ is the probability that execution of intention τ_i will require the shared resource res_u.

The scheduling algorithm is shown in Algorithm 7.1 - 7.3. The set S contains all feasible partial schedules (initially a single empty schedule). The set of candidate intentions is processed in descending order of priority. A candidate intention τ_i is added to a partial schedule from S if it can be assigned a start time s_i such that the probability of the intention completing successfully before its deadline is at least α and where the probability that there are no resource conflicts with currently scheduled intentions is at least β. Each candidate intention τ_i is initially assigned a start time of 'now', and then an attempt is made to schedule τ_i at the cycle immediately after the expected end time of each currently scheduled intention. These steps are repeated for rest of partial schedules in S.

The scheduling algorithm tries out all possible combinations of parallel and sequential scheduling and determines all feasible start times for the intention τ_i and updates the set S accordingly. Intentions, which are not feasible in the context of any of the current partial schedules, are dropped. When all candidate intentions have been processed, the scheduler returns the first feasible instance of the schedule. The resulting schedule is 'maximally feasible' (no more intentions can be added to the schedule if the scheduled intentions are to remain feasible at the specified execution time and resource conflict confidence levels) and moreover, intentions which are dropped are incompatible with some scheduled higher priority intention(s).

Theorem 5. *The Algorithm 7.1- 7.3 worse-case complexity* $O(n \cdot e^n)$.

Algorithm 7.1 AgentSpeak^{MT}(RT) Scheduling Algorithm (Part 1)

```
function SCHEDULE(I)
    S := {∅}
    for all τ ∈ I in descending order of priority do
        S' := ∅
        for all Γ ∈ S do
            s := now
            S' := S' ∪ {SCHEDULE-SERIES(s, τ, Γ)}
            for all (s', τ') ∈ Γ do
                s := s'
                S' := S' ∪ {SCHEDULE-PARALLEL(s, τ, Γ)}
                s := s' + et(τ', α) − ex(τ') + δ_c
                S' := S' ∪ {SCHEDULE-SERIES(s, τ, Γ)}
            end for
        end for
        if S' ≠ ∅ then
            S := S'
        end if
    end for
    return first(S)
end function
function SCHEDULE-PARALLEL(s, τ, Γ)
    if FEASIBLE(Γ ∪ {(s, τ)}) then
        return Γ ∪ {(s, τ)}
    else
        return ∅
    end if
end function
```

Proof. In the worst case, when priority varies with deadline and intentions can be inserted into the schedule in any order (i.e., in parallel or in sequence), then the

Algorithm 7.2 AgentSpeak^{MT}(RT) Scheduling Algorithm (Part 2)

```
function SCHEDULE-SERIES(s, τ, Γ)
    Γ' := ∅
    for all (s', τ') ∈ Γ do
        if s' < s then
            Γ' := Γ' ∪ {(s', τ')}
        else
            s' := s' + et(τ, α) − ex(τ) + δ_c
            Γ' := Γ' ∪ {(s', τ')}
        end if
    end for
    if FEASIBLE(Γ' ∪ {(s, τ)}) then
        return Γ' ∪ {(s, τ)}
    else
        return ∅
    end if
end function
```

feasibility of each new intention involves checking the feasibility of all possible partial schedules of intentions. A number of all possible partial schedules is defined by an ordered Bell number. Hence, the Algorithm 7.1 - 7.3 has a worse-case complexity $O(F_{|I|})$, where F_n is a Fubini number or an ordered Bell number, given by

$$F_n = \sum_{k=0}^{\infty} \frac{k^n}{2^{k+1}} \tag{7.3}$$

Berend and Tassa in [2] established the following weak bound for the Bell numbers:

$$F_n < \left(\frac{0.792n}{\ln(n+1)} \right)^n. \tag{7.4}$$

We can transform the right side of the Equation 7.4 as follows:

$$\lim_{n\to\infty} \frac{0.792n}{\ln(n+1)} < \lim_{n\to\infty} \frac{(0.792n)'}{(\ln(n+1))'} = \lim_{n\to\infty} \frac{0.792}{\frac{1}{n+1}} < \lim_{n\to\infty} n. \tag{7.5}$$

Algorithm 7.3 AgentSpeak^{MT}(RT) Scheduling Algorithm (Part 3)

```
function FEASIBLE(Γ)
    for all (s, τ) ∈ Γ do
        if s + et(τ, α) − ex(τ) > d(τ) then
            return false
        end if
        p := 1
        for all (s', τ') ∈ Γ \ {(s, τ)} do
            if s < s' + et(τ', α) ∧ s' < s + et(τ, α) then
                for k from 1 to n do
                    p := p × (1 − P(τ, res_k) × P(τ', res_k))
                    if p < β then
                        return false
                    end if
                end for
            end if
        end for
    end for
    return true
end function
```

Then combining Equation 7.4 and 7.5, we can derive the following bound on ordered Bell numbers:

$$F_n < n^n \rightarrow F_n < e^{n \cdot \ln(n)}. \tag{7.6}$$

The complexity of the scheduling algorithm is then $O(e^{n \cdot \ln n})$ (or $O(n \cdot e^n)$). □

The logarithmic growth is the inverse of exponential growth and is very slow, hence the algorithm complexity is higher than exponentials, but it is lower than double exponential i.e., $e^n < e^{n \cdot \ln n} < e^{n^2}$. The complexity of the algorithm is very high for a real-time system. However in the special case of one resource, which is

used with probability 0 or 1, the schedule can be computed in polynomial time. We will consider this special case below.

7.3.2 The AgentSpeak^{MT}(RT) Interpreter

After scheduling, an AgentSpeak(RT) agent starts executing intentions with start time $s = now$. The execution of an intention differs from the single-tasking version of AgentSpeak(RT). Each intention can be in one of two states: executing and executable. An intention is *executable* if the first step in the topmost plan in the stack of partially executed plans that forms the intention is a goal or an action which is not currently executing (i.e., the action has either completed executing or has yet to begin execution). If the first step in an executable intention is an action which has completed execution, the completed action is removed. If the completed action was the last step in a plan, the completed plan is popped from the stack of partially executed plans that forms the intention and any shared resources required solely for the execution of the plan are released. Execution then proceeds from the next step of the topmost plan in the intention. Executing an executable intention involves executing the first goal or action of the body of the topmost plan in the stack of partially executed plans which forms the intention. Executing an achievement goal adds a corresponding internal goal addition event and removes the achievement goal from the body of the plan. Executing a test goal involves finding a unifying substitution for the goal and the agents base beliefs. If a substitution is found, the test goal is removed from the body of the plan and the substitution is applied to rest of the body of plan. If no such substitution exists, the intention is dropped and removed from the schedule. Executing an action results in the invocation of the Java code that implements the action and changes the state of the intention from executable to

executing. We assume that action execution is performed in a separate thread, and execution of the AgentSpeak^{MT}(RT) interpreter resumes immediately after initiating the action.

7.3.3 Atomic Intentions

In this section we consider the special case of an agent with only one shared resource which is used by each plan with probability 0 or 1, i.e., for all τ : $P(\tau, res) =$ 0 or 1. In this case there are some intentions that can be scheduled in parallel with other intentions at $s = now$, and some intentions that must be scheduled in series (i.e., atomic intentions).

A schedule $\{(s_1, \tau_1), \ldots, (s_n, \tau_n)\}$ is feasible if each intention will complete execution before its deadline with probability α. More precisely, a set of intentions $\{(s_1, \tau_1), \ldots, (s_n, \tau_n)\}$ is feasible if

1. for each scheduled intention (s_i, τ_i)

$$s_i + et(\tau_i, \alpha) - ex(\tau_i) \leq d(\tau_i), \tag{7.7}$$

2. the probability that there is no resource conflict with intentions $\{(s_j, \tau_j) \mid s_i < s_j + et(\tau_j, \alpha) \wedge s_j < s_i + et(\tau_i, \alpha)\}$ which are already scheduled and execute concurrently with τ_i is equal to 1:

$$\prod_{\tau_j} (1 - P(\tau_i, res) \times P(\tau_j, res)) = 1. \tag{7.8}$$

The scheduling algorithm is shown in Algorithm 7.4. The set of candidate intentions is processed in descending order of priority. A candidate intention τ, which requires shared resource (i.e., $P(\tau, res) = 1$) is added to the schedule if it can be inserted into the schedule in deadline order while meeting its own and all currently

scheduled deadlines. On the other hand, a candidate intention that does not require the resource (i.e., $P(\tau, res) = 0$) is scheduled at $s = now$. Intentions which are not feasible in the context of the current schedule are dropped. The resulting schedule is computed in polynomial time (in fact, quadratic time) in the size of the set I, and will be priority-maximal (no more intentions can be added to the schedule if the scheduled intentions are to remain feasible at the specified confidence level) and intentions which are dropped are incompatible with some scheduled higher priority intention(s).

Single resource multitasking in AgentSpeak(RT) is similar to capabilities provided by atomic plans in Jason [6] and 2APL [11]. An atomic plan is a plan which should be executed ensuring that its execution is not interleaved with the execution of the goals and actions of other plans of the same agent. The resulting AgentSpeak^{MT}(RT) agent system is more expressive than Jason and 2APL in one sense, as Jason and 2APL cannot run non-atomic plans in parallel with an atomic one. However, it is less expressive in another sense, as in Jason and 2APL a non-atomic plan can have an atomic subplan.

7.4 Example

In this section we demonstrate the advantages of AgentSpeak^{MT}(RT) over AgentSpeak(RT) by showing how it improves the behaviour of the intelligent agent that we developed to control a nuclear power plant in Chapter 5.

Recall that the agent performs diagnostics, provides the required electric power level, maintains the plant's parameters, and handles unexpected situations. The agent has access to seven power plant resources: the reactor of the power plant res_1, turbines res_2, the main power line res_3, the steam generator res_4, the electric

Algorithm 7.4 AgentSpeak^{MT}(RT) Scheduling Algorithm (Intermediate Case)

```
function SCHEDULE(I)
    Γ_s := ∅
    Γ_p := ∅
    for all τ ∈ I in descending order of priority do
        if P(τ, res) = 0 then
            if Γ_p ∪ {(now, τ)} is feasible then
                Γ_p = Γ_p ∪ {(now, τ)}
            end if
        else
            s = now
            Γ'_s = ∅
            for all (s', τ') ∈ Γ_s do
                if d(τ') ≤ d(τ) then
                    Γ'_s := Γ'_s ∪ {(s', τ')}
                    s := s' + et(τ', α) − ex(τ')
                else
                    s' := s' + et(τ, α) − ex(τ)
                    Γ'_s := Γ'_s ∪ {(s', τ')}
                end if
            end for
            if Γ'_s ∪ {(s, τ)} is feasible then
                Γ_s = Γ'_s ∪ {(s, τ)}
            end if
        end if
    end for
    return Γ_p ∪ Γ_s
end function
```

generator res_5, the feed water pump res_6, and the condenser water pump res_7. Note that the execution time profile of the plans is the same as before.

The agent has two plans for diagnostics of the power plant's systems. The `full-diagnostics` plan requires all seven resources for execution, while the `express-diagnostics` plan requires access only to the reactor, turbines, steam generator, and the electric generator. Hence the execution resource profiles associated with plans `full-diagnostics` and `express-diagnostics` are $\langle res_1, 1;\ res_2, 1;\ res_3, 1;\ res_4, 1;\ res_5, 1;\ res_6, 1;\ res_7, 1\rangle$ and $\langle res_1, 1;\ res_2, 1;\ res_3, 0;\ res_4, 1;\ res_5, 1;\ res_6, 0;\ res_7, 0\rangle$.

```
Plans:
  @full-diagnostics
  +!diagnostics : full-diagnostics
        :: res1,res2,res3,res4,res5,res6,res7 <-
      check(reactor);
      check(power-line);
      check(turbines);
      check(steam-generator);
      check(elec-generator);
      check(feed-water-pump);
      check(cond-water-pump).

  @express-diagnostics
  +!diagnostics : express-diagnostics
        :: res1;res2;res4;res5 <-
      check(reactor);
      check(turbines);
      check(steam-generator);
      check(elec-generator).
```

In order to change the water level, the agent has to run the feed water pump.

The water control plans therefore require only resource res_6 (feed water pump) for their execution, i.e., the execution resource profile is $\langle res_1, 0;\ res_2, 0;\ res_3, 0;\ res_4, 0;\ res_5, 0;\ res_6, 1;\ res_7, 0\rangle$.

```
@water-control
+!set-water-level(L) : water-level(60)
      :: res6 <-
    change-water-level(L).

@norm-water-level
+!norm-water-level   : water-level-setpoint(L)
      :: res6 <-
    change-water-level(L).
```

However changing the power level of the power plant involves control of the reactor, the steam generator, the electric generator, and the condenser water pump. The `power-control` plan therefore requires resources res_1, res_4, res_5, res_7, i.e., the execution resource profile is $\langle res_1, 1;\ res_2, 0;\ res_3, 0;\ res_4, 1;\ res_5, 1;\ res_6, 0;\ res_7, 1\rangle$.

```
@power-control
+!set-power-level(K) : true
      :: res1; res4; res5; res7 <-
    change-power-level(K).
```

Finally the emergency protection plan that executes emergency stop procedures will obviously need to control all power plan's systems. As a result, the plan requires all resources for its execution, i.e., the execution resource profile is $\langle res_1, 1;\ res_2, 1;\ res_3, 1;\ res_4, 1;\ res_5, 1;\ res_6, 1;\ res_7, 1\rangle$.

```
@emergency-protection
```

```
+short-circuit : true
      :: res1;res2;res3;res4;res5;res6;res7 <-
    ?power-level(P)
    emergency-stop;
    !set-power-level(0);
    .print(''Main line power cut'');
    reset-plant;
    !set-power-level(P).
```

As in the previous example (see Section 5.7) the AgentSpeak^{MT}(RT) agent has to handle four events: a perform diagnostics `+!diagnostics` goal, a change feed water level `+!set-water-level` goal, a change power production `+!set-power-level` goal, and a short circuit event `+short-circuit`. Events are matched against the plan library. Plans `full-diagnostics`, `water-control`, `power-control` and `emergency-protection` are executable and (individually) feasible. They are added to the set of candidate intentions. Recall that the intention that corresponds to `+short-circuit` event, has the highest priority (25), whereas the intentions to set power level and feed water level have medium priority (10), and the intention to perform the diagnostics has the lowest priority (4).

The scheduling algorithm (Algorithm 7.1–7.3) attempts to schedule the candidate intentions in descending order of priority. As previously, the intention that corresponds to the `+short-circuit` event is inserted into the schedule, which is currently empty. The intention to increase water level cannot be scheduled in parallel with the `short-circuit` intention due to the resource conflicts. Although it can be scheduled sequentially in deadline order.

In contrast to the previous example (Section 5.7), the algorithm schedules the intention to change the power level in parallel with the intention to increase the

water level. The intention `diagnostics` has a resource conflict with the scheduled intentions and cannot be scheduled in parallel with them. However, the agent is now able to schedule it sequentially in deadline order.

We can see that, compared to previous example, the agent now can increase the water level and change the power level at the same time, and complete diagnostics by the deadline.

7.5 Real-Time Agency

In this section we show that under certain assumptions, an AgentSpeak^{MT}(RT) agent provides real-time guarantees (which we believe are reasonable for real-time applications and discuss them below). We prove that the time required to execute a single cycle of the AgentSpeak^{MT}(RT) interpreter (and hence the reactivity delay of the agent) is bounded. We also show that an AgentSpeak^{MT}(RT) agent commits to a priority-maximal set of intentions, and that, given a fixed schedule, the probability that an intention will complete successfully by its deadline is α and the probability that there will be no resource conflict with other scheduled intentions is at least β.

We make the following assumptions about the agent's program and task environment:

1. the set of possible beliefs has a fixed maximal size (for example, the set of possible beliefs can be restricted to the set of ground instances of any literal appearing in a belief context or a test goal for a finite set of constants);

2. the set of possible goals has a fixed maximal size (for example, the set of possible goals can be limited to the set of ground instances of any atomic formula appearing in an achievement goal for a finite set of constants);

3. the maximal possible deadline of any event is a constant d_{max} (relative to the current time);

4. the minimal expected execution time for any plan is a constant t_{min};

5. there is a maximal expected execution time, t_{max}, for any action in the agent program (i.e., $t_{max} = max(et(a, \alpha))$ for any action a at the specified α);

6. the time required to execute a single cycle of the interpreter is small relative to the minimal expected execution time of any action in the agent's program[1], and

7. there is a minimal probability P_{min} that a plan requires at least one shared resource.

Theorem 6. *If the sets of possible beliefs and goals, the maximal expected action execution time and the maximal deadline have a fixed maximal size, and the minimal plan execution time has a fixed minimal size and the minimal probability that a plan requires at least one shared resource has a fixed minimal value, then the time required to execute a single cycle of the AgentSpeak*$^{\mathrm{MT}}$*(RT) interpreter is bounded by a constant* δ_c.

Proof. Recall that the current interpreter is similar to the single-tasking version, but functions *shed* and *exec* are different.

As already mentioned in Section 7.3 the complexity of the scheduling algorithm is $O(|I| \cdot e^{|I|})$. $|I|$ is bounded if the maximal possible deadline of any event is a constant d_{max} (assumption 3), the minimal expected execution time for any plan is a constant

[1]The minimal expected execution time of any agent action has to be at least double that of the time required to execute a single interpreter cycle. Otherwise the agent will not be able to execute intentions concurrently.

t_{min} (assumption 4) and the minimal probability that a plan requires at least one resource is a constant P_{min} (assumption 7). Then the maximal possible number of schedulable intentions is bounded by $d_{max}/t_{min} \times n \times (1-\beta)/(1-P_{min}^2)$, where n is the number of shared resources. If we assume that the time required to add an achievement goal to the set of events, evaluate a goal query, and initiate the execution of an action are bounded by a constant, then the time to compute *exec* is bounded by a constant.

As *evt* and *opt* are unchanged, the time required to execute a single cycle of the AgentSpeak^{MT}(RT) interpreter is therefore bounded by a constant δ_c. □

It therefore follows that the *reactivity delay* of an AgentSpeak^{MT}(RT) agent (the time required for an agent to recognise or become aware of changes in its environment) is also bounded.

Theorem 7. *The reactivity delay of an AgentSpeak^{MT}(RT) agent is bounded.*

Proof. Similarly to single-tasking AgentSpeak(RT), the maximum reactivity delay is for an event which arrives just after the evaluation of *evt* begins, and is bounded by $2\delta_c$. □

Next we prove that the multitasking scheduling algorithm returns a priority-maximal set of intentions (as defined in Chapter 6).

Theorem 8. *Given a partially ordered set of intentions $I = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where $p(\tau_i) \geq p(\tau_j)$ for $i < j$, the scheduling algorithm generates a priority-maximal set of intentions $\Gamma \subseteq I$.*

Proof. We can see that, as for the single-tasking version, the scheduling algorithm generates a sequence of sets starting with $\Gamma_{0,0} = \emptyset$, $S_0 = \{\emptyset\}$ and for all sets $\Gamma_{i,j} \in S_i$, $\Gamma_{i,j} = \Gamma_{i-1,j} \cup \{(s_i, \tau_i)\}$, $\tau_i \in I$ if $\Gamma_{i-1,j} \cup \{(s_i, \tau_i)\}$ is feasible, otherwise $S_i = S_{i-1}$. The

set $\Gamma_{n,1} \in S_n$ is Γ. By construction, Γ is a feasible set of intentions. Γ is also clearly a maximally feasible subset of I: there is no $\tau \in I$ such that $\tau \notin \Gamma$ and $\Gamma \cup \{\tau\}$ is feasible. To prove that it is priority-maximal, let $\tau_i \in I$, $\{(s_i, \tau_i)\}$ be feasible, and $(s_i, \tau_i) \notin \Gamma$. We need to show that τ_i is incompatible with some subset of Γ, which contains only intentions of the same or higher priority than $p(\tau_i)$. Since the intentions are added to Γ in descending order of priority, when τ_i is considered and found incompatible with $\Gamma_{i-1,j}$, $p(\tau_i) \leq \min(\{p(\tau') : \tau' \in \Gamma_{i-1,j}\})$. □

Theorem 9. *The probability that an intention τ will be executed by its deadline successfully in a static environment is equal to α.*

Proof. From the fact that (1) δ_c is small relative to the minimal expected execution time of any action in the agent's program (assumption 6), and (2) the execution time profiles of the plans provides us the estimate of duration of task with probability α, an intention τ will complete execution before its deadline in a static environment with a probability α. □

Theorem 10. *The probability that the will be no resource conflict between an intention τ and another scheduled intention in a static environment is $\geq \beta$.*

Proof. Immediate, from the feasibility requirement in the scheduling algorithm. The start time of an intention τ is chosen such that the probability that the intentions with which τ executes concurrently will not require any of the shared resources required by τ is at least β. □

7.6 Dynamic Environments

We can modify the model of the 'difficulty' of the agent's environment from Section 6.2 to incorporate parallel execution of intentions, in order to prove that the

AgentSpeak^{MT}(RT) architecture with multitasking provides real-time guarantees for a dynamic environment. We show how this model can be used to determine the priority of intentions which can be reliably scheduled in an environment of specified difficulty, and to estimate the probability that an intention of given priority will not be displaced from the schedule by the arrival of an intention of higher priority.

Recall that we characterise a task environment in terms of the average arrival rate and time available for the execution of intentions of a given priority. Let intentions of priority i be associated with an average triggering rate r_i, and an average time available for their execution a_i, such that $a_i \geq t_i$ where t_i is the average execution time of intentions of priority i at the specified confidence level α. We assume that the amount of uncommitted or 'slack' time unused by intentions of priority i is $s_i \geq 0$ for all i given and otherwise empty schedule. We also assume that intentions of priority i are associated with an execution resource profile $\langle P(\tau_i, res_1), \ldots, P(\tau_i, res_n)\rangle$. We assume that intentions with the same priority level have the same resource requirements and cannot be executed concurrently with each other. In contrast to the previous model of the 'difficulty' of the agent's environment, intentions may be scheduled concurrently with other intentions, and as a result, it may be possible to execute more intentions by their deadlines with probability α.

In the worst case the schedule is full, intentions complete their execution just before their deadlines, and the probability β that an intention does not compete for resources with other scheduled intentions is equal to 1. The probability that a candidate intention of priority i will be schedulable is equal to a probability that the intention can be scheduled either in series (as in single-tasking version of AgentSpeak(RT)) or in parallel with other intentions of priority $> i$. As these are disjoint alternatives, we can consider these probabilities separately. We have derived the probability that the intention τ of priority i will be scheduled serially F_s in Chapter

6 (see Equation (6.4)).

For intentions of priority i to be concurrently schedulable there must no resource conflicts with intentions of priority greater than i. If the maximum priority of any intention is m, then the probability $F_p(\tau_i)$ that the intention τ_i will be schedulable in parallel with intentions of priority $j > i$, is given by

$$F_p(\tau_i) = \prod_{\tau_j} \prod_{u=1}^{n} (1 - P(\tau_i, res_u) \times P(\tau_j, res_u)) \tag{7.9}$$

We now can determine the probability $F(\tau_i)$ that the intention τ_i is schedulable. Assuming the arrival of intentions is a Poisson process[2], the probability that τ_i is schedulable, $F(\tau_i)$, is given by

$$F(\tau_i) = F_s(\tau_i) + F_p(\tau_i) - F_s(\tau_i) \times F_p(\tau_i) \tag{7.10}$$

We can also determine probability $U(\tau_i)$ that a scheduled intention of priority i is displaced from the schedule by the arrival of a higher priority intention.

Recall that in the case of the single-tasking version of AgentSpeak(RT), for a scheduled intention of priority $i < m$ to be displaced, sufficient intentions of priority $> i$, with total execution time $> s_i$, must arrive within a time interval a_i (Chapter 6). For multi-tasking execution we also have to take into account fact that intentions of priority i may be scheduled concurrently with other intentions of priority $> i$, and as a result may not be displaced by the arrival of these intentions. A set of intentions with priorities $i + 1, i + 2 \ldots, m$ sufficient to displace an intention of priority i can be represented as a vector $\langle u_{i+1}, \ldots, u_m \rangle$ where $u_i \in \{i + 1, \ldots, m\}$ is the number of intentions of priority i which arrive within a_i. To displace an intention of priority i such vectors must satisfy a number of conditions:

[2]Poisson process is one of the most important models used in queueing theory. It allows to determine the probability of a number of events occurring in a fixed period of time if these events occur with a known average rate and independently of the time since the last event.

1. the number of intentions of each priority must be feasible given s_i;
2. the combined execution time of all intentions in the set must be greater than s_i;
3. the combined execution time of the intentions should exceed s_i by at most the least execution time of any intention in the set;

In general, a scheduled intention τ_i of priority $i < m$ will be displaced if the uncommitted time s_i is less than total execution time of an arrived sequence of intentions of priority $> i$, and τ_i can't be scheduled in parallel with these intentions and each other.

The probability that an intention of priority i is displaced, $U(\tau_i)$, is the probability that at least the number of intentions of each priority level specified by one such sequence occurs. We can extend Equation 6.5 to the AgentSpeak^{MT}(RT) agent, the probability, $U(\tau_i)$, is then

$$U(\tau_i) = \left(1 - \prod_{\langle u_{i+1},\ldots,u_m\rangle} (1 - U(u_{i+1}) \times \ldots \times U(u_m))\right) \times \prod_{k=i+1}^{m} (1 - F_p(\tau_k)). \quad (7.11)$$

where $U(u_i) = \left(1 - \sum_{x=0}^{u_i-1} \frac{e^{-\lambda_i}\lambda_i^x}{x!}\right)$ is the joint probability that at least u_i intentions are added to the schedule in time a_i, where λ_i is an average number of intentions of priority i.

The probability that an AgentSpeak^{MT}(RT) agent will execute an intention of priority j to completion is then

$$E(\tau_i) = F(\tau_i) \times (1 - U(\tau_i)) \times \alpha. \quad (7.12)$$

For given values of r_i, a_i and t_i, we can therefore determine the probability that an intention of given priority will not be scheduled, or will be displaced from the schedule by a higher priority intention before it can complete successfully.

7.7 Summary

In this chapter we have described how we can extend the AgentSpeak(RT) architecture to allow the parallel execution of intentions. We have discussed the multitasking approach for parallel execution of intentions and proved real-time properties in that case. We have also extended a model of the 'dynamism' of the agent's environment by allowing parallel execution of intentions, and shown how it can be used to determine the priority of intentions which can be reliably scheduled, and estimated the probability that a scheduled intention of given priority will be successfully executed by its deadline. In the next chapter we will discuss the characterisation of real-time task environments.

Chapter 8

AgentSpeak(RT) Environment Characterisation

In this chapter we present a characterisation of real-time task environments for an agent, and describe how it relates to AgentSpeak(RT) execution time profiles for a plan and an action. We also explore the relationship between the estimation accuracy of the execution time of a plan and the syntactic complexity of the agent program.

8.1 Classification of real-time task environments

In this Section we describe an environment characterisation related to the estimated execution time of a single action or a plan. The range of different task environments for a real-time agent is obviously very wide. We identify a small number of dimensions on which real-time task environments can be categorised and describe typical execution time profile for each dimension. Recall that the *execution time profile* is a function of execution time which specifies the probability that the action or the plan will terminate successfully, and in AgentSpeak(RT) it is specified as a power

function with three parameters k, ρ, α_{max} as follows

$$\alpha(t_e) = f_\phi(t_e, k, \rho, \alpha_{max}) = \begin{cases} \left(\frac{t_e}{\rho}\right)^k & t_e \leq \rho \\ \alpha_{max} & t_e > \rho \end{cases}$$

where t_e is the execution time of a plan; the parameter k defines the inflection of the execution time profile curve, ρ is a scale parameter, and α_{max} is the maximal possible level of confidence in a particular agent environment.

There are three dimensions which can be used to categorise a real-time task environment for an agent: *difficulty*, *confidence* and *error*. These dimensions in AgentSpeak(RT) are determined by parameters of the execution time profile above: k, ρ, α_{max}.

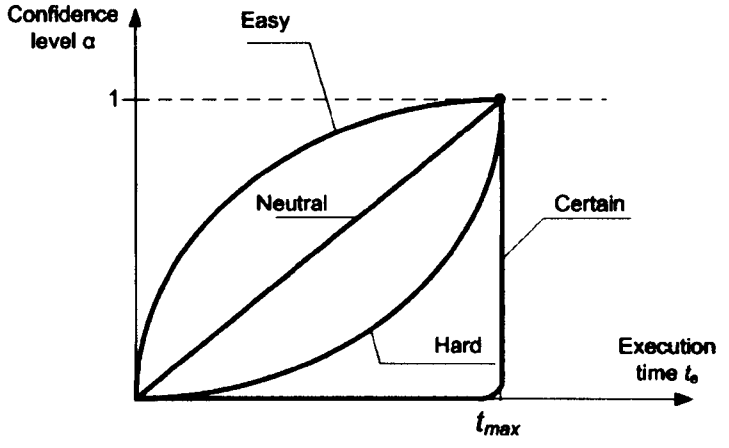

Figure 8.1: Difficulty of real-time task environments

Difficulty. The difficulty of an environment is determined by the maximum execution time of an action or a plan t_{max} (i.e., the time required to complete an action or a plan with α_{max} confidence) in the environment and the type of the execution time profile (i.e., shape of the execution time profile). The maximum execution time of the execution time profile $f_\phi(t_e, k, \rho, \alpha_{max})$ is defined by the

parameter ρ, and the type of the profile is defined by the parameter k. We can distinguish four types of the real-time agent environment based on this dimension: *easy, neutral, hard, certain.* If an agent can successfully complete an intention before its deadline with high confidence level and relatively short execution time, then the environment is called 'easy'. An 'easy' environment is described by the function f_ϕ with $0 < k < 1$. On the other hand an environment is called 'hard' if an agent has to spend a significant amount of time to successfully accomplish an intention. A 'hard' environment is described by the function f_ϕ with $k > 1$. For example, an empty corridor can be considered as easy real-time task environment for a delivery robot, because it is very likely that it can successfully complete a delivery. A crowded corridor with many obstacles is a hard environment for the same agent because it has to spend significantly more time to successfully complete a task. 'Neutral' real-time task environments are characterised by linear dependence of the confidence level and the execution time. Neutral environments are specified by the function f_ϕ with $k = 1$. Another real-time task environment type is a 'certain' environment, where an agent can successfully complete an intention with the maximum probability and spending only a fixed amount of time. This kind of environments is characterised by the function f_ϕ with very large values of parameter k (in ideal case $k \to \infty$). The maximum execution time of an action or a plan $t_{max} = \rho$ in an environment is also a very important parameter of the environment's difficulty. In environments characterised by a small value of ρ, it takes less time to successfully execute an action (with a given level of confidence) than in an environment where ρ is large. Profiles for all described types of environment are presented in Figure 8.1.

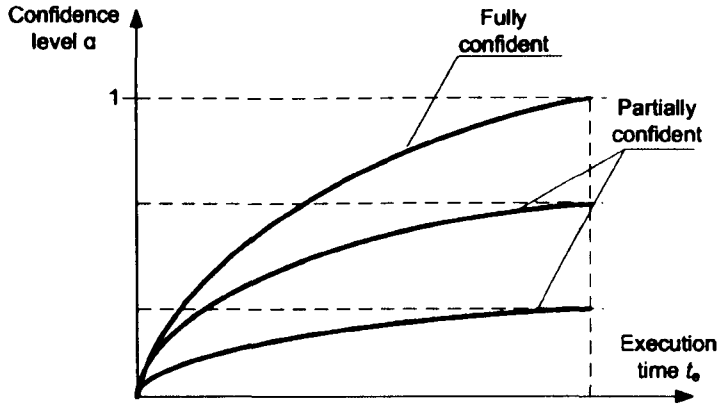

Figure 8.2: Confidence of real-time task environments

Confidence. If the maximum possible confidence level α_{max} for an environment is equal to 1, then we say that a real-time task environment is fully confident, otherwise it is partially confident. In principle, an agent can achieve its goals with any required level of confidence in a fully confident environment. If the environment is partially confident, then an agent can't guarantee that an intention will be successfully completed by the deadline with probability more than α_{max} (see Fig. 8.2). For example, flipping a fair coin is clearly a partially confident environment, while walking from one place to another is fully confident.

Error. Estimation error is another important characteristic of a real-time task environment. If the agent developer has full information about the agent's environment, he can accurately estimate the execution time profile and its parameters k, ρ, α_{max}. On the other hand if the agent developer has only partial knowledge about environment then the estimated execution time profile will have some deviation ε_f from real execution time profile i.e., $f_{est}(t_e, k^{est}, \rho^{est}, \alpha^{est}_{max}) = f_{real}(t_e, k, \rho, \alpha_{max}) + \varepsilon_f$ (See Figure 8.3). Therefore the parameters k, ρ, α_{max} of the estimated execution time profile will have measurement errors i.e., $k^{est} =$

$k + \varepsilon_k$, $\rho^{est} = \rho + \varepsilon_\rho$, $\alpha_{max}^{est} = \alpha_{max} + \varepsilon_\alpha$. These errors ε_k, ε_ρ, ε_α depend on our knowledge about the environment.

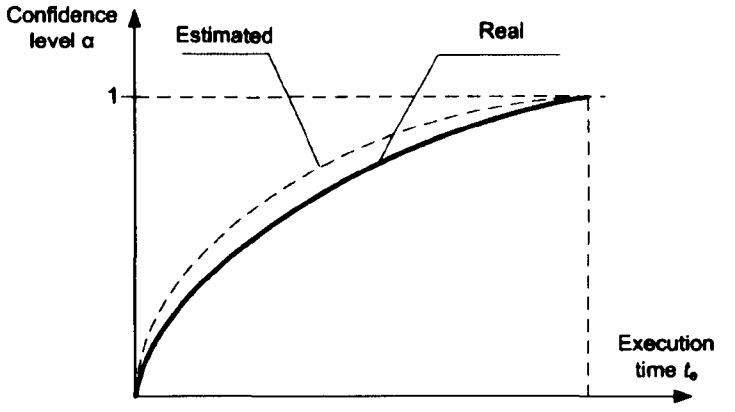

Figure 8.3: Error of real-time task environments

8.2 Accuracy of Execution Time Estimation

In the previous section, we have described how we can categorise real-time task environments. We have shown that estimating the execution time of a plan depends on the type of the real-time task environment. In this Section we will investigate how the syntactic complexity of an agent program (i.e., availability of subgoals, loops, etc.) influences the expected execution time a plan. We also show how to compute the estimation error for an agent program of a different syntactic complexity.

Intuitively we can say the higher syntactic complexity of an agent program, the greater the estimation error in the plan execution time and the less certain we can be regarding the execution time of the plan.

As noted above, in many cases an agent developer will be unable to obtain precise execution time profiles for an agent's actions $\langle a_1, a_2, \ldots \rangle$, and the execution times for actions will include estimation errors $\langle \varepsilon(a_1), \varepsilon(a_2), \ldots \rangle$ (e.g., $t_e(a_1) = 5 \pm 0.5$).

If we assume a simple agent programming language, which allows only sequences of primitive actions within a plan, e.g., $\pi = \{a_1, a_2, \ldots, a_n\}$, then the expected execution time $t_e(\pi)$ of a plan π is equal to

$$t_e(\pi) = et(\pi, \alpha) \pm \varepsilon(\pi) = \sum_{i=1}^{n} et(a_i, \alpha) \pm \sum_{i=1}^{n} \varepsilon(a_i) \tag{8.1}$$

That is, the estimation error for the execution time of the plan π is equal to the sum of estimation errors $\varepsilon(a_i)$ for each action a_i in the plan. Assuming that all actions have equal estimation errors $\varepsilon(a)$ we can determine the estimation error for the plan π as follows

$$\varepsilon(\pi) = \sum_{i=1}^{n} \varepsilon(a_i) = n \cdot \varepsilon(a). \tag{8.2}$$

Although the expected execution time of the plan deviates from the real execution time, for relatively small estimation errors $\varepsilon(a_i)$ we can determine the expected execution time of a plan with high confidence. However, this agent language can be used to solve only very simple tasks.

If the agent programming language supports test subgoals and actions (i.e., the agent's plan has a general form as follows $\pi = \{a_1, \ldots, a_k, ?g_{k+1}, a_{k+2}, \ldots, a_r, ?g_{r+1}, a_{r+2}, \ldots, a_n\}$, then the expected execution time $t_e(\pi)$ of the plan π will also depend on the result of evaluating the test goals $?g_{k+1}, ?g_{r+1}, \ldots$. Obviously if one test goal fails during execution then the execution time of the plan π will be less than if the goal succeeds. This means that the expected execution time $t_e(\pi)$ depends on the probabilities $\varphi(?g_k)$ that test goals $?g_k$ do not fail during their execution.

The expected execution time $t_e(\pi)$ including an estimation error $\varepsilon(\pi)$ of the plan

π is as follows

$$t_e(\pi) = et(\pi,\alpha) \pm \varepsilon(\pi) = \left(\sum_{i=1}^{k} et(a_i,\alpha) \pm \sum_{i=1}^{k} \varepsilon(a_i)\right) \cdot (1 - \varphi(?g_{k+1}))+$$
$$+ \left(\sum_{i=1}^{r} et(a_i,\alpha) \pm \sum_{i=1}^{r} \varepsilon(a_i)\right) \cdot (1 - \varphi(?g_{r+1})) \times \varphi(?g_{k+1}) + \ldots$$
$$\ldots + \left(\sum_{i=1}^{n} et(a_i,\alpha) \pm \sum_{i=1}^{n} \varepsilon(a_i)\right) \cdot \prod_{\forall j} \varphi(?g_j) \pm \varepsilon_\varphi(\pi) \quad (8.3)$$

or

$$t_e(\pi) = et(\pi,\alpha) \pm \varepsilon(\pi) =$$
$$= \sum_{\forall s_j \in S} \left[\left(\sum_{i=1}^{s_j} et(a_i,\alpha) \pm \sum_{i=1}^{s_j} \varepsilon(a_i)\right) \cdot (1 - \varphi(?g_{s_j})) \prod_{l=s_1}^{s_{j-1}} \varphi(?g_l)\right] \pm \varepsilon_\varphi(\pi) \quad (8.4)$$

where $S = \{s_1, s_2, \ldots\}$ are the indices of the test goals in the plan and $\varepsilon_\varphi(\pi)$ is the estimation error caused by uncertainty of test goals

$$\varepsilon_\varphi(\pi) = 1 - \frac{\sum_{\forall s_j \in S} \left[\left(\sum_{i=1}^{s_j} et(a_i,\alpha)\right) \cdot (1 - \varphi(?g_{s_j})) \prod_{l=s_1}^{s_{j-1}} \varphi(?g_l)\right]}{\sum_{i=1}^{n} et(a_i,\alpha)}$$

We can see that the total estimation error of the plan execution time in that case is nonlinear and strongly depends on the results of evaluating the tests.

Let's assume again for simplicity that all actions a_i have equal estimation error $\varepsilon(a)$ and all tests goals $?g_k$ have equal probability $\varphi(?g)$ of being successfully executed, then the estimation error of the plan π duration has the following general form

$$\varepsilon(\pi) = b \cdot \varepsilon(a) + \left(1 - \left[\sum_{i=0}^{\nu-1} c_i(1 - \varphi(?g))\varphi^i(?g) + \varphi^\nu(?g)\right]\right) \quad (8.5)$$

$$b = \sum_{i=0}^{\nu-1} \left[(1 - \varphi(g))\varphi^i(g) \cdot (s_{i+1} - 1)\right] + n \cdot \varphi^\nu(g),$$

where $c_i = \dfrac{\sum_{j=1}^{s_i} et(a_j, \alpha)}{\sum_{j=1}^{n} et(a_j, \alpha)}$ is a constant which equals to the ratio of the sum of the execution time of each action before the test goal $?g_{s_i}$ to the total execution time of all actions in the plan π, and ν is the number of test goals within the plan π.

However if an agent programming language supports achieve goals and primitive actions in plans, then the estimation error in the expected execution time of a plan depends on the estimation errors in expected execution times of plans π_j invoked by subgoals in π. Let the number of plans to achieve a subgoal $!g_k$ be n_k, and the probability that 'subplan' π_j will be triggered by a goal $!g_k$ be $P(!g_k, \pi_j)$.

Then the expected execution time of a plan $\pi = \{a_1, \ldots, a_k, !g_{k+1}, \ldots, a_r, \ldots, a_n\}$ is given by

$$t_e(\pi) = et(\pi, \alpha) \pm \varepsilon(\pi) = \sum_{i=1}^{k} et(a_i, \alpha) \pm \sum_{i=1}^{k} \varepsilon(a_i) + t_e(!g_{k+1}) + \ldots$$
$$\ldots + \sum_{i=r}^{n} et(a_i, \alpha) \pm \sum_{i=r}^{n} \varepsilon(a_i) \quad (8.6)$$

where

$$t_e(!g_k) = et(!g_k, \alpha) \pm \varepsilon(!g_k) =$$
$$= \sum_{j=1}^{n_k} (et(\pi_j, \alpha) \times P(!g_k, \pi_j)) \pm \left(\sum_{j=1}^{n_k} (\varepsilon(\pi_j) \times P(!g_k, \pi_j)) + \varepsilon_g(!g_k) \right)$$

$$\varepsilon_g(!g_k) = 1 - \frac{\sum_{j=1}^{n_k} (et(\pi_j, \alpha) \times P(!g_k, \pi_j))}{\max_{1 \le j \le n_k} et(\pi_j, \alpha)}$$

Assuming, that the execution time of any action in the plan can be estimated with equal estimation error $\varepsilon(a)$, each subgoal $!g_k$ has same number of related plans n_k and all plans are equally likely (i.e., have the same probability $P = \frac{1}{n_k}$) to be

chosen to accomplishing the subgoal, the estimation error $\varepsilon(\pi)$ is given by

$$\varepsilon(\pi) = n \cdot \varepsilon(a) + \mu \cdot P \cdot \sum_{j=1}^{n_k} \varepsilon(\pi_j) + \mu \cdot (1 - \theta \cdot P) =$$

$$= n \cdot \varepsilon(a) + \mu \cdot \frac{1}{n_k} \cdot \sum_{j=1}^{n_k} \varepsilon(\pi_j) + \mu \cdot \left(1 - \theta \cdot \frac{1}{n_k}\right) \quad (8.7)$$

where

$$\theta = \frac{\sum_{j=1}^{n_k} et(\pi_j, \alpha)}{\max\limits_{1 \le j \le n_k} et(\pi_j, \alpha)},$$

μ is the number of subgoals within plan π.

Recall that modern programming languages like AgentSpeak(L), etc., allow both achieve and test goals together with primitive actions. Hence we can now combine equations (8.4), (8.6) and determine how the expected execution time of a plan π, written in a AgentSpeak(L)-like agent programming language, depends on the availability of primitive actions, achieve and test goals. Assuming that the general form of the plan is $\pi = \{a_1, \ldots, a_k, !g_{k+1}, \ldots, a_r, ?g_{r+1}, a_{r+2}, \ldots, a_n\}$ the execution time is described by the following equation

$$t_e(\pi) = et(\pi, \alpha) \pm \varepsilon(\pi) = \sum_{\forall s_j \in S} \left[\left(\left(\sum_{i=1}^{s_j} et(a_i, \alpha) + \sum_{\forall k:\ k < s_j} et(!g_k, \alpha) \right) \pm \right.\right.$$

$$\left.\left. \pm \left(\sum_{i=1}^{s_j} \varepsilon(a_i) + \sum_{\forall k:\ k < s_j} \varepsilon(!g_k) \right) \right) \times (1 - \varphi(?g_{s_j})) \prod_{l=s_1}^{s_{j-1}} \varphi(?g_l) \right] \pm \varepsilon_{asl}(\pi) \quad (8.8)$$

$$\varepsilon_{asl}(\pi) = 1 - \frac{\sum\limits_{\forall s_j \in S} \left[\left(\sum\limits_{i=1}^{s_j} et(a_i \mid !g_i, \alpha) \right) \cdot (1 - \varphi(?g_{s_j})) \prod\limits_{l=s_1}^{s_{j-1}} \varphi(?g_l) \right]}{\sum\limits_{i=1}^{n} et(a_i \mid !g_i, \alpha)}$$

The estimation error for the plan π is given by following equation

$$\varepsilon(\pi) = b \cdot (\varepsilon(a) + \varepsilon(!g)) + \left(1 - \left[\sum_{i=0}^{\nu-1} c_i^* (1 - \varphi(?g))\varphi^i(?g) + \varphi^\nu(?g)\right]\right) \quad (8.9)$$

where

$$c_i^* = \frac{\sum_{j=1}^{s_i} et(a_j \mid !g_j, \alpha)}{\sum_{j=1}^{n} et(a_j \mid !g_j, \alpha)}$$

Although the estimation error (Equation 8.9) for AgentSpeak(L)-like agent languages is higher than the estimation error for the simple agent language case (Equation 8.2), a high confidence in the estimated execution time of a plan can still be achieved with small estimation errors $\varepsilon(a_i)$ by careful utilization of test and achieve goals (e.g., by minimising the use of test goals and related plans for each achieve goal within an agent program).

Further complication of the language syntax by adding loops (like in PRS and 2/3APL) may significantly increase the estimation error. These syntax structures may include different combination of tests and subgoals, hence the resulting estimation error is equal to the estimation error for combinations of subgoals and tests. Also the agent developer has to estimate the probability that a particular loop will successfully terminate as a function of execution time (i.e., it requires execution time profile). However it is very hard to accurately compute this profile, especially in case when the end condition is some event. This means that the estimation error for a plan may become unacceptably high. The consequences of the high estimation error are that an agent may be unable to schedule some of the feasible intentions, as the expected execution time of an intention can be significantly higher than the real execution time, and unable to complete intentions, which are 'false-feasible' (i.e., intentions that are infeasible, are considered to be feasible because of the estimation error).

There is a trade-off in real-time agent systems between the estimation accuracy of the estimated execution time of an intention and the syntactic complexity of the agent program. An agent developer should carefully choose of the syntactic complexity of

an agent program keeping in mind the fact that it may affect the estimation accuracy of the execution time of an agent intention.

8.3 Summary

In this chapter we have presented a characterisation of real-time agent environments, and shown how different classes of environment affect AgentSpeak(RT) execution time profiles for plans and primitive actions. We have also shown how the estimation accuracy for the expected execution time of a plan depends on the syntactic complexity of an agent program (i.e., what syntax constructs are allowed in the agent language).

Chapter 9

Conclusions and Future Work

"The main lesson of thirty-five years of AI research is that the hard problems are easy and the easy problems are hard. The mental abilities of a four-year-old that we take for granted - recognizing a face, lifting a pencil, walking across a room, answering a question - in fact solve some of the hardest engineering problems ever conceived..."

Steven Pinker

9.1 Conclusions

9.1.1 Contributions

This thesis looked at the problem of real-time guarantees in 'Belief-Desire-Intention' high-level declarative agent programming languages. We have reviewed existing agent architectures in Chapter 3. These architectures have several features and tools such as metalevel reasoning, time-outs, and plan repair rules, which allow agents situated in dynamic real-time environments to meet real-time constraints. However program-

ming real-time BDI agents in these BDI languages is hard as they have to be adjusted for every particular application. This is the problem that this research is trying to address.

We have provided a definition of what it means for BDI agents to operate in real-time, or to satisfy real-time guarantees in Chapter 2. We have defined a *real-time BDI agent* as one which schedules the execution of its intentions so as to respond to events by their deadlines; if not all events can be processed by their deadlines, the agent favours intentions responding to high priority events. For a real-time BDI agent, correctness of the agent's program depends not only on the actions the agent performs but also on the time at which it performs them.

The AgentSpeak(RT) architecture presented in Chapter 5 provides a flexible framework for the development of real-time BDI agents. An AgentSpeak(RT) agent pursues a priority-maximal set of intentions which can be achieved by their deadlines with a specified confidence level. If not all intentions can be achieved by their deadlines, the agent prefers intentions with greater priority.

By varying the level of confidence, the developer can control the degree of 'optimism' the agent adopts when determining the time required to complete a task in a given environment. Higher levels of confidence will typically result in the agent allowing more time to complete a task, and cause fewer tasks to be scheduled in a given period of time. As tasks are scheduled in priority order, increasing the level of confidence required also has the effect of causing the agent to focus more on high priority tasks at the expense of lower priority tasks which might be achievable given a more optimistic view of execution time. If deadlines and priorities are not specified for top-level goals or belief invoked plans, the behaviour of the agent defaults to that of a non real-time BDI agent. Real-time goals and tasks triggered by changes in the agent's beliefs can be freely mixed with tasks for which no deadline and/or priority

has been specified by the developer or user. Tasks without deadlines will be processed after any task with a specified deadline, and for tasks with the same deadline, the agent will prefer tasks of higher priority.

In Chapter 6 we proved that the guaranteed reaction time of the AgentSpeak(RT) interpreter is bounded and we proved probabilistic guarantees of successful execution of intentions in a static environment. We have developed a simple model of the 'difficulty' of the agent's environment, and showed how this model can be used to determine the priority of intentions which can be reliably scheduled in an environment of specified difficulty, and to estimate the probability that a scheduled intention of a given priority will be displaced from the schedule by the arrival of an intention of higher priority and will be accomplished by the specified deadline.

In Chapter 7 we have extended the AgentSpeak(RT) architecture to allow the parallel execution of intentions through multitasking and showed that the real-time properties of the AgentSpeak(RT) architecture still hold under parallel execution of intentions. The AgentSpeak^{MT}(RT) scheduling algorithm is unlikely to be feasible for more than a small number of resources and/or intentions. However the special case in which there is a single resource is tractable. This case is similar to the notion of "atomic plans" found in other BDI-based agent programming languages, and greatly extends the applicability of AgentSpeak(RT). We have extended the model of the 'difficulty' of the real-time environment by allowing parallel execution of intentions and used it to derive a probability that in a dynamic environment a scheduled intention of a given priority will be successfully executed by its deadline.

Finally, we have listed a number of dimensions along which task environments can be categorised, and described typical execution time profiles for each dimension. We have also described a relationship between the estimated execution time of plans in an agent's environment and the syntactic complexity of agent programs.

9.1.2 Possible Applications

While the focus in this thesis is to provide real-time guarantees for a generic BDI agent we can identify a number of potential application domains. The areas of application of real-time BDI agent systems is very wide and includes management and control of air traffic systems, high-level robotic control, telecommunications, business processes, financial systems etc. The example of a high-level intelligent control system for a nuclear power plant was presented in Chapter 5. We consider some application domains below.

High-Level Robotic Control

A high-level robotic control system allows a robot to perform user tasks in environments without continuous human guidance (i.e., it allows a robot to operate autonomously). High-level intelligent control is widely used in military, industrial and airspace systems, such as UAVs, Mars rovers, autopilots etc.

Figure 9.1: Mars Exploration Rover

One example of autonomous robotic system is a Mars exploration rover. The Mars rover in Fig. 9.1 is a six-wheeled, solar-powered robot. Due to the distance

communication with the Earth base is only possible few times a day. Therefore, real-time intelligent control is required to image, navigate, sample, find the position of the sun, return to the lander, etc.

In each communication session the agent is given a set of goals to achieve, each of which specifies either navigation to a specific location or collection of samples. Also the agent perceives the environment in order to detect rover position, obstacles etc. There is a deadline for each event provided by mission duration, planets position, daily energy limitations, and priorities are assigned to events based on the importance of the requested operation, e.g., safety of the rover is more important than any mission task. The agent, for example, may have following events

```
+!navigate(rover,space2)[6pm,10]
+battery(rover,20)[1pm,20]
```

indicates the acquisition of a goal to navigate the rover to space2 with deadline 6pm and priority 10, and a new belief that the battery level is 20%; it should be noted that placing itself in sunny positions has higher priority than the goal to navigate the rover to space2.

Once the agent receives the required tasks, it checks the current position and status of its subsystems, and navigates to the required location. If the deadline for a task has passed, the agent drops it.

Financial and Business Systems

The significant increase of acquiring and utilising financial monitoring systems in today's financial institutions such as risk management systems, trading systems etc. The serious challenge is that these systems need to process a big amount of different financial information in real-time. We believe that real-time intelligent agents are well

suited to dealing with the problem of monitoring and processing dynamic information in a financial sector.

Consider, for example, a trading agent which buys commodities in an marketplace. The agent receives requests from clients to bid on their behalf, and notifications of goods for sale which the agent may also bid for on its own behalf. The market operates as a series of concurrent first-price sealed-bid auctions of short duration in which sellers offer goods for sale.

The trading agent responds to two kinds of events: requests from clients to make a specified bid on their behalf in a particular auction, and notifications of new auctions where the agent may decide to bid on its own behalf. The deadline of an event is the deadline for bids for the corresponding auction, and priorities are assigned to events based on the importance of the client (for client requests) and the type good sold in the auction (for auction notifications). The agent's primary role is as a broker, so the priority of auction notification events is lower than that of client requests. For example, the events

```
+!bid(client2, a102, price2)[1010, 15]
+auction(a201, good1)[1060, 10]
```

indicates the acquisition of a goal to bid price2 on behalf of client2 in auction a102 with deadline 1010 and priority 15, and a new belief that good1 is being offered in auction a201, with deadline 1060 and priority 10.

When the agent receives a request to bid in an auction, it checks that the requesting agent is a client and that the client has sufficient credit before making the bid. When it receives notification of a new auction, the agent may decide to bid on its own account. Determining what price it should offer depends on the type of good offered for sale. Once the deadline for an auction has passed, the market determines

the highest bid and notifies successful agents of their purchase and remaining credit level.

9.1.3 Limitations

The research presented in this thesis has a number of limitations where there is room for improvement.

The main limitation of the AgentSpeak(RT) agent framework is that the performance of the agent depends on the accuracy of execution time profiles for each plan. High estimation errors may cause unexpected agent behaviour (e.g., agent may be either unable to schedule some of the intentions, which are feasible and schedulable, or unable to complete intentions, which are 'false-feasible') and affect real-time guarantees.

Although the AgentSpeak(RT) architecture provides flexible predictable framework, the expressive power of the agent language syntax is limited. The language does not allow internal belief addition (i.e., originated from inside the agent's plan) nor goal deletion events.

Moreover the AgentSpeak(RT) language does not support if-tests and loops syntax structures as additional information is required. However conditionals and loops can be simulated in AgentSpeak using subgoals and plans.

9.2 Future Work

There are many ways in which this research can be extended. However, there are four specific areas which at this stage show significant promise. These include:

Handling Intention Failures. In the thesis we have assumed that a failed intention would be dropped and removed from an agent's intention structure. However,

in many cases more flexible approaches to plan failure are required. In future work we plan to investigate alternative approaches to handling plan failure.

Scheduling Enhancement. The current AgentSpeak(RT) scheduling algorithm (Algorithm 5.1) distinguishes between agent's tasks only based on priority, deadline, and execution time. However these criteria may not be enough for some applications, such as e-commerce, logistics etc. The agent has to reason about the trade-offs of different courses of actions in these applications. It would be interesting to try to extend the current scheduling algorithm to handle one extra criterion, e.g., the expected cost of plan and action execution.

Environment Classification. In this thesis we have identified a very small number of dimensions along which real-time task environments can be classified in Chapter 8. However, the range of different types of real-time task environments is obviously very high. We plan to provide explicit classification of real-time task environments.

Real-Time MAS. The AgentSpeak(RT) agent described in this thesis was largely stand-alone. While the single agent can normally operate in a real-time environment, the AgentSpeak(RT) architecture can not provide real-time guarantees for a multi-agent system (MAS). MAS is composed of multiple intelligent agents which interact to achieve common goals (team goals) or to achieve personal goals (individual goals). In both cases the real-time grantees for these goals depends on time critical behaviour of each individual agent (i.e., the agent has to be real-time) and real-time interaction between these agents. We aim to define real-time guarantees for a multi-agent system and develop real-time protocols for interaction between agents as part of future work.

9.3 List of Dissemination

All original contributions of the thesis are listed below.

- *The ARTS Real-Time Agent Architecture.* We presented a new BDI architecture, ARTS, which allows the development of agents that guarantee (soft) real-time performance. ARTS extends ideas from PRS and JAM to include goals and plans which have deadlines and priorities, and schedules intentions so as to achieve a priority-maximal set of intentions by their deadlines. The paper has been published in the Lecture Notes in Computer Science: Languages, Methodologies, and Development Tools for Multi-Agent Systems [36].

- *Agent programming with priorities and deadlines.* We presented AgentSpeak(RT), a real-time BDI agent programming language based on AgentSpeak(L). AgentSpeak(RT) extends AgentSpeak intentions with *deadlines* and *priorities*. The AgentSpeak(RT) interpreter commits to a priority-maximal set of intentions. We prove real-time properties of the language. The paper has been accepted in Proceedings of the Tenth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'11).

- *AgentSpeak(RT): a Real-Time Agent Programming Language.* We presented a multitasking approach to the parallel execution of intentions in the AgentSpeak(RT) architecture. We also proved real-time properties of AgentSpeak^{MT}(RT), a multitasking version of AgentSpeak(RT) architecture. The paper is being prepared for submission in Proceedings of the Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS).

Bibliography

[1] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[2] D. Berend and T. Tassa. Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 1, 2010.

[3] R. Bordini, A. Bazzan, R. de, O. Jannone, D. Basso, R. Vicari, and V. Lesser. AgentSpeak(XL): Efficient Intention Selection in BDI Agents via Decision-Theoretic Task Scheduling. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems AAMAS'02*, pages 1294–1302, New York, NY, USA, 2002. ACM.

[4] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak using Jason.* Wiley, 2007.

[5] R. H. Bordini and J. F. Hübner. BDI Agent Programming in AgentSpeak Using Jason. *Lecture Notes in Computer Science*, 1:143–164, 2006.

[6] R. H. Bordini, J. F. Hübner, and R. Vieira. *Multi-agent programming : languages, platforms and applications*, chapter Jason and the Golden Fleece of agent-oriented programming, pages 3–37. Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, New Yor, USA, 2005.

[7] V. Botti, C. Carrascosa, V. Julian, J. Soler, and I. Computacin. The ARTIS Agent Architecture: Modelling Agents in Hard Real-Time Environments. In *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW99)*, pages 63–76. Springer-Verlag, 1999.

[8] M. Bratman. Two faces of intention. *The Philosophical Review*, 93(3):375–405, July 1984.

[9] C. Carrascosa, J. Bajo, V. Julian, J. M. Corchado, and V. Botti. Hybrid multi-agent architecture as a real-time problem-solving model. *Expert Systems Applications*, 34(1):2–17, 2008.

[10] J. Chakareski, J. Apostolopoulos, and B. Girod. Low-complexity rate-distortion optimized video streaming. In *Proceedings of the International Conference on Image Processing (ICIP)*, volume 3, pages 2055–2058, Oct. 2004.

[11] M. Dastani, D. Hobo, and J.-J. C. Meyer. Practical Extensions in Agent Programming Languages. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '07)*, pages 1–3, New York, NY, USA, 2007. ACM.

[12] K. S. Decker and V. R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 2:215234, 1993.

[13] S. G. Deshpande. High quality video streaming using content-awareadaptive frame scheduling with explicit deadlineadjustment. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 777–780, New York, NY, USA, 2008. ACM.

[14] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI89))*, Detroit, Michigan, 1989.

[15] M. P. Georgeff and A. L. Lansky. Procedural Knowledge. In *Proceedings of the IEEE (Special Issue on Knowledge Representation)*, volume 74, pages 1383–1398. IEEE Press, 1986.

[16] J. S. Gu and C. W. de Silva. Development and implementation of a real-time open-architecture control system for industrial robot systems. *Engineering Applications of Artificial Intelligence*, 17(5):469 – 483, 2004.

[17] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357 – 401, 1999.

[18] M. J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of The Third International Conference on Autonomous Agents*, pages 236–243, Seattle, WA, 1999.

[19] J. F. Hübner, R. H. Bordini, and M. Wooldridge. Programming declarative goals using plan patterns. In M. Baldoni and U. Endriss, editors, *Declarative Agent Languages and Technologies IV, 4th International Workshop, DALT 2006, Hakodate, Japan, May 8, 2006, Selected, Revised and Invited Papers*, volume 4327 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2006.

[20] T. Konnerth, B. Hirsch, and S. Albayrak. Jadl - an agent description language for smart agents. In M. Baldoni and U. Endriss, editors, *DALT*, volume 4327 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2006.

[21] J. Lee and E. H. Durfee. Structured circuit semantics for reactive plan execution systems. In *Proceedings of the twelfth national conference on Artificial Intelligence (AAAI'94)*, volume 2, pages 1232–1237, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[22] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[23] J. D. C. Little. A proof of the queueing formula L = λ W. *Operations Research*, 9:383–387, 1961.

[24] R. Machado and R. Bordini. Running AgentSpeak(L) Agents on SIM_AGENT. In J.-J. Meyer and M. Tambe, editors, *Intelligent Agents VIII Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, Lecture Notes in Artificial Intelligence, pages 158 – 174, Seattle, WA, August 2002. Springer-Verlag.

[25] D. Morley and K. Myers. The SPARK Agent Framework. In *Proc. of the Third Int. Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS-04)*, pages 712–719, New York, NY, July 2004.

[26] K. L. Myers. *PRS-CL: A Procedural Reasoning System. User's Guide.* SRI International, Center, Menlo Park, CA, March 2001.

[27] K. L. Myers and D. E. Wilkins. The Act Formalism. Technical report, SRI International Artificial Intelligence Center, Menlo Park, CA, September 1997.

[28] A. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world*, pages 42–55, 1996.

[29] A. S. Rao and M. P. Georgeff. BDI-agents: From Theory to Practice. In *Proceedings of the First Intl. Conference on Multiagent Systems (ICMAS'95)*, San Francisco, 1995.

[30] P. Shaw and R. Bordini. Towards alternative approaches to reasoning about goals. In *Proceedings of Declarative agent languages and technologies V: 5th international workshop (DALT 2007)*, volume 4897/2008, pages 104–121, Honolulu, HI, USA, May 14 2007. Springer-Verlag New York Inc, Springer.

[31] P. H. Shaw, B. Farwer, and R. H. Bordini. Theoretical and experimental results on the goal-plan tree problem. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1379–1382, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[32] J. Thangarajah and L. Padgham. An empirical evaluation of reasoning about resource conflicts. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '04)*, pages 1298–1299, Washington, DC, USA, 2004. IEEE Computer Society.

[33] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting & avoiding interference between goals in intelligent agents. In *Proceedings of the 18th international joint conference on Artificial intelligence (IJCAI'03)*, pages 721–726, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[34] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting & exploiting positive goal interaction in intelligent agents. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems (AAMAS '03)*, pages 401–408, New York, NY, USA, 2003. ACM.

[35] J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In *Proceedings of the 15th European Conference on Artifical Intelligence 2002 (ECAI 2002)*, pages 18–22. IOS Press, 2002.

[36] K. Vikhorev, N. Alechina, and B. Logan. The ARTS Real-Time Agent Architecture. In M. Dastani, A. El Fallah Segrouchni, J. Leite, and P. Torroni, editors, *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, volume 6039 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, Turin, Italy, September 2010.

[37] R. Vincent, B. Horling, V. Lesser, and T. Wagner. Implementing soft real-time agent control. In *Proceedings of the fifth international conference on Autonomous agents (AGENTS '01)*, pages 355–362, New York, NY, USA, 2001. ACM.

[38] T. Wagner, A. Garvey, and V. Lesser. Criteria-Directed Heuristic Task Scheduling. *International Journal of Approximate Reasoning*, 19:91–118, Jyly 1998.

[39] H. Wang and C. Wang. APACS: a Multi-Agent System with Repository Support. *Knowledge-Based Systems*, 9(5):329 – 337, 1996.

[40] H. Wang and C. Wang. Intelligent agents in the nuclear industry. *Computer*, 30(11):28 –31, November 1997.

[41] D. E. Wilkins and K. L. Myers. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, 5(6):731–761, 1995.

[42] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, *In Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, pages 470–481, Toulouse, France, April 2002. Morgan Kaufmann.

[43] M. Wooldrige and N. Jennings. Intelligent agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):1–62, June 1995.

Appendix A

AgentSpeak(RT): Implementation

In this chapter we discuss specific implementation details of the AgentSpeak(RT) architecture. Specifically we present a complete AgentSpeak(RT) language BNF. Also we show the interpreter code and present a GUI for the agent language interpreter.

A.1 Overview

AgentSpeak(RT) is an intelligent agent architecture that combines the best aspects of several leading-edge agent theories and intelligent agent frameworks. AgentSpeak(RT) is influenced by: Belief-Desire-Intention(BDI) theories [8], Procedural Reasoning System (PRS) [15], SRI International's PRS-CL [26], The Act Formalism [27], JAM [18], Jason [6], etc.

The AgentSpeak(RT) implementation is based on the JAM source code. We have implemented AgentSpeak(RT) in Java; the current prototype implementation includes the core language described above and implementations of some basic actions. Additional user-defined actions can be added using a Java API. Actions are implemented as Java methods. AgentSpeak(RT) supports two mechanisms for defining

primitive actions: writing a class which implements the `ExternalAction` interface, and direct invocation of methods in existing Java legacy code.

A.2 AgentSpeak(RT) BNF Grammar

agent	::=	*belief-base init-goals plans*
belief-base	::=	(*belief* ".")*
init-goals	::=	(*goal* ".")*
plans	::=	(*plan*)*
goal	::=	"!" *literal* | "?" *literal*
belief	::=	*literal*
plan-spec	::=	*plan time-profile* [*resource-profile*]
plan	::=	"@" *plan-name event* [":" *context*] ["::" *resources*] "<-" *body* "."
plan-name	::=	**string**
event	::=	"+" ["!"] *literal* | "-" *literal*
context	::=	*true* | *literal* ("&" *literal*)*
body	::=	*true* | *step* (";" *step*)*
step	::=	*action* | *goal*
action	::=	[".")]**a** ["(" *term-list* ")"]
time-profile	::=	"time-profile" ":" *number*"," *number*"," *number*
resource-profile	::=	"resource-profile" ":" (resource"," *number*";")*
resources	::=	*resource*("," *resource*)*
resource	::=	**r** ["(" *term-list* ")"]
function	::=	**f** "(" *term-list* ")"
literal	::=	["not"] *atomic-formula*
atomic-formula	::=	**p** ["(" *term-list* ")"]
term-list	::=	*term* ("," *term*)*
term	::=	*constant* | *variable* | *function*
constant	::=	**integer** | **float** | **string**
number	::=	**integer** | **float**
variable	::=	**VAR**

where **p**, **f**, **a**, and **r** are respectively a predicate, a functor, an action and a resource symbols; **VAR** is a variable name.

A.3 Primitive Actions

Primitive actions are the basic operations an agent can perform to change its environment in order to achieve its goals. AgentSpeak(RT) distribution contains several internal primitive actions, but agent developers can specify additional functionality using the `ExternalAction` interface. We will cover each of these two types of primitive actions in more detail below.

A.3.1 User-defined Actions

User-defined primitive actions are implemented as Java methods. AgentSpeak(RT) supports two mechanisms for defining primitive actions: writing a Java class which implements the `ExternalAction` interface, and direct invocation of methods in existing legacy Java code.

The `ExternalAction` interface has a following form:

```
public interface ExternalAction
{
    public Boolean execute(String name, ExpList args,
                           Binding binding, Goal currentGoal);
}
```

The execute method's arguments are:

name is a string which specifies the primitive action name,

args is a list of arguments of the primitive action passed to it from the agent's plan,

binding is a structure which holds the plan variable bindings associated with the passed-in arguments,

currentGoal is a field which holds the goal of the plan which is invoking this primitive action.

An agent developer has full access to all arguments of the primitive action using the *args* and *binding* arguments. Each primitive action returns a boolean value. The `TRUE` value indicates successful completion of an action. The `FALSE` value indicates failure of the action.

For example, the action `change-water-level(L)` will cause the agent to change the feed water level for the reactor.

```
//
// Change a level of feeding water.
//
public Boolean execute(String name, ExpList args,
                       Binding binding, Goal currentGoal)
{
  if (args.getSize() != 1) {
    System.err.println(''Invalid number of arguments: '' + args.getSize() +
                                    +'' to function'' + name);
    return FALSE;
  }
  float init-vol          = getVol();    // get current water volume
  float des-vol           = args.pop();
```

```
    // Technical implementation
    try {
        while (des-vol != getVol()){
            if (des-vol < init-vol)
                pump.run();   // normal mode
            else if (des-vol > init-vol)
                pump.rev();  // reverse mode
        }
        pump.stop();
        return TRUE;
    }
    catch (Exception e) {
      System.err.println("AgentSpeak(RT)::Cannot run pump : " + e);
      return FALSE;
    }
  }
```

A.3.2 Internal Primitive Actions

This section describes predefined internal primitive actions available in the AgentSpeak(RT) distribution.

.drop-intention removes an intention I from an intention structure.

.drop-intention(literal: I).

Example: `.drop-intention(remove(paper,space2))`: removes the intention that was triggered by the "`!remove(paper,space2)`" event.

.drop-all-intentions removes all intentions from an agent's intention structure.

.drop-all-intentions.

Example: `.drop-all-intentions`.

.exec execute external program.

.exec(String commandString),

where commandString is a string containing an executable program.

Example: `.exec`("C:\Program Files\Skype\Phone\Skype.exe").

.fail causes plan failure.

.fail.

Example: `.fail`.

.getAPL gets a description of the list of all applicable plans in an interpreter cycle.

.getAPLplans.

Example: `.getAPLplans`.

.getCurrentIntention returns the description of the current intention.

.getCurrentIntention.

Example: `.getCurrentIntention`.

.getTime gets the current time.

.getTime(int T).

Example: `.getTime(Time)`.

.print permits output to the standard output device (i.e., System.out).

.print(Term* M).

The input for this action is a list of any valid terms.

Example: `.print("Performing next action")`.

.printBeliefs prints agent's belief base.

.printBeliefs.

Example: `.printBeliefs.`

.printIntentions prints agent's intention structure.

.printIntentions.

Example: `.printIntentions.`

.send sends a message to an agent.

.send(Term-list receiver, String ilf, Term* message)

The input arguments are the *receiver* of the message, the *illocutionary force* of the message, and *message content.*

Example: `.send(agent1, tell, apple(red)).`

.succeed makes the plan finish successfully.

.succeed.

Example: `.succeed.`

.wait suspends the intention for the specified time.

.wait(Integer T).

Example: `.wait(58).`

A.4 The Interpreter

We have discussed the reasoning cycle of AgentSpeak(RT) in Chapter 5. This Section describes the interpreter code. The interpreter code is shown in Algorithm A.1.

The *schedule* function takes the set of intention I and returns the schedule for an agent. The functions *head* and *body* return the head and body of an intended

Algorithm A.1 AgentSpeak(RT) Interpreter Cycle

$B, E := B, E \cup P, G$
for all $\langle e, \tau \rangle \in E$ **do**
  $O_e := \{\pi\theta \mid \theta$ is an applicable unifier for e and plan $\pi\}$
  $\pi\theta := S_O(O_e)$
  if $\pi\theta \neq \emptyset$ **and** $\tau \notin I$ **then**
    $I := I \cup \pi\theta$
  else if $\pi\theta \neq \emptyset$ **and** $\tau \in I$ **then**
    $I := (I \setminus \tau) \cup push(\pi\theta\sigma, \tau)$ where σ is an mgu for $\pi\theta$ and τ
  else if $\pi\theta = \emptyset$ **and** $\tau \in I$ **then**
    $I := I \setminus \tau$
  end if
end for
$I :=$ SCHEDULE(I)
if $I \neq \emptyset$ **then**
  $\tau := first(I)$, $\pi := pop(\tau)$
  if $first(body(\pi)) = !g(t_1, \ldots, t_n)$ **then**
    $push(head(\pi) \leftarrow rest(body(\pi)), \tau)$
    $E = \{\langle +!g(t_1, \ldots, t_n), \tau \rangle\}$
  else if $first(body(\pi)) = ?g(t_1, \ldots, t_n)$ **then**
    if $?g(t_1, \ldots, t_n)\theta$ is an answer substitution **then**
      $push(head(\pi)\theta \leftarrow rest(body(\pi))\theta, \tau)$
    else
      $push(\pi, \tau)$
    end if
  else if $first(body(\pi)) = a(t_1, \ldots, t_n)$ **then**
    if $execute(a(t_1, \ldots, t_n), et(a(t_1, \ldots, t_n), \alpha))$ **then**
      $push(head(\pi) \leftarrow rest(body(\pi)), \tau)$
    else
      $push(\pi, \tau)$
    end if
  end if
end if

plan, and *first* and *rest* are used to return the first and all but the first elements of a sequence. The function *push* takes a plan (and any substitution) and an intention and pushes the plan onto the top of the intention. The function *pop* removes and

returns the topmost plan of an intention. The function *execute* takes an action and an expected execution time, and executes the action for at most the expected execution time. It returns true if the action completes successfully within its expected execution time; otherwise it returns false.

A.5 AgentSpeak(RT) Interface

In general, AgentSpeak(RT) is run using a graphical interface that is layered on top of the jEdit text editor. jEdit is a cross platform programmer's text editor written in Java that is customizable with plugins. AgentSpeak(RT) plugin is based on the Jason platform plugin [4]. The interface is depicted in Figure A.1.

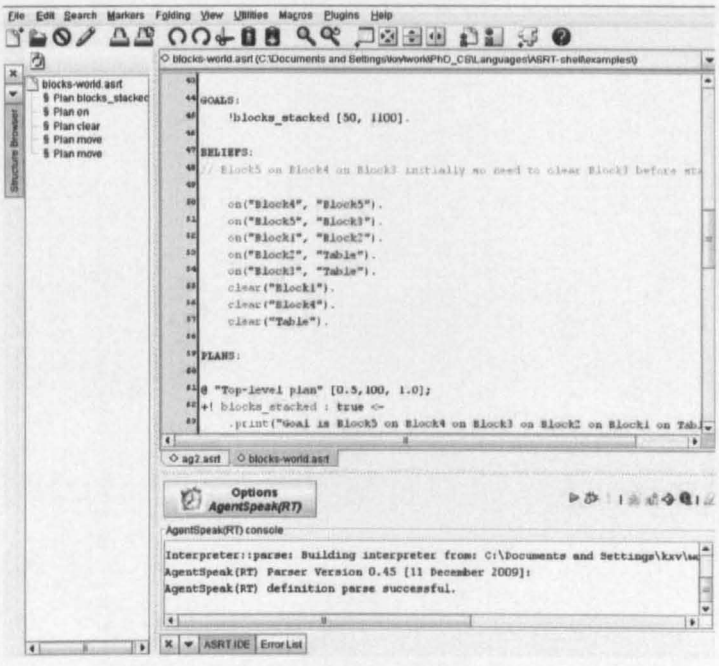

Figure A.1: The AgentSpeak(RT) interface

The interface is divided into several graphical areas. Each area presents different

kinds of information which is used to control and interact with the AgentSpeak(RT) agent. The largest area in the AgentSpeak(RT) interface is the agent program area which is used by user to program an agent, primitive actions or environments. This area has syntax highlighting and multiple tabs. The AgentSpeak(RT) plugin provides templates for an agent program, a primitive action and an environment.

The left side area represents a *Structure Browser*. This area displays the name of the agent program and its structure: initial beliefs and goals, plan names etc. It facilitates navigation in the agent program.

The control area at the bottom of the program area provides instruments to run and to debug an agent. In addition, it allows adding agents, environment and internal actions to the project, and displays error messages.