TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents

Innes Andrew Ferguson

Clare Hall

A dissertation submitted for the degree of Doctor of Philosophy in the University of Cambridge

October 1992

Summary

The computer-controlled operating environments at such facilities as automated factories, nuclear power plants, telecommunications centres, and space stations are continually becoming more complex. As this complexity grows, it will be increasingly difficult to control such environments with centralised management and scheduling policies that are both robust in the face of unexpected events and flexible at dealing with operational and environmental changes that might occur over time. One solution to this problem which has growing appeal is to distribute control of such operations to a number of intelligent, task-achieving computational agents.

Real-world domains are likely to be populated by multiple agents. In such domains agents will typically perform a number of complex tasks requiring some degree of attention to be paid to environmental change, temporal constraints, computational resource bounds, and the impact the agents' shorter term actions might have on their longer term goals. Operating in the real world means having to deal with unexpected events at several levels of granularity — both in time and space. While agents must remain reactive in order to survive, some amount of strategic and predictive decision making will be required if agents are to coordinate their actions with other agents and handle complex tasks in an effective manner.

This dissertation presents a new integrated agent architecture, designed to provide rational, autonomous, mobile agents with the diverse range of behaviours normally required to carry out complex, resource-constrained tasks in dynamic, real-time, multi-agent domains. Upon surveying a collection of existing architectures and after due consideration of the requirements for producing effective, robust, and flexible behaviours in a particular class of such domains, the resulting software control architecture — the TouringMachine agent architecture — has been designed through integrating a number of deliberative and non-deliberative control functions. Arranged in a layered fashion, the combination of these functions endows agents with a rich collection of reactive, goal-oriented, reflective, and predictive capabilities.

In recognition of the complex relationship which exists between an agent's internal configuration, its task environment, and its ensuing behavioural repertoire, the agent architecture has been implemented in conjunction with a feature-rich, instrumented simulation testbed. The testbed, which permits the creation of a diverse set of single- and multi-agent navigation task scenarios, has been used to evaluate the utility of the architecture and to identify some of its main strengths and weaknesses.

TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents

Innes A. Ferguson

Declaration

I hereby declare that this dissertation is the result of my own work and, unless explicitly stated in the text, contains nothing which is an outcome of work done in collaboration. No part of this dissertation has already been or is currently being submitted for any degree, diploma or other qualification at any other university.

Innes A. Ferguson
October, 1992

Related Publications

Selected aspects of the research described in this dissertation, as well as some earlier related work have been documented or published elsewhere:

I.A. Ferguson. TouringMachines: Autonomous Agents with Attitudes. *IEEE Computer*, 25(5), May, 1992.

I.A. Ferguson. TouringMachines: Autonomous Agents with Attitudes. Technical Report 250, Computer Laboratory, University of Cambridge, UK, April, 1992. Will also appear in J. Herath (ed.) *Readings in Computer Architectures for Intelligent Systems*, IEEE Computer Society Press: Los Alamitos, CA. To be published in 1993.

I.A. Ferguson. Toward an Architecture for Adaptive, Rational, Mobile Agents. In E. Werner and Y. Demazeau (eds.) *Decentralized AI 3*, Elsevier Science (North Holland): Amsterdam, 1992. Also appears in *Proc. of the Third European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds*, Kaiserslautern, Germany, August, 1991.

I.A. Ferguson. Toward Intelligent Robotic Control. *AI for Engineers*, 5:3, AI Applications Institute, University of Edinburgh, Scotland, UK, September, 1991.

I.A. Ferguson. Toward an Architecture for Adaptive, Rational, Mobile Agents (extended abstract). In M.P. Georgeff and A.S. Rao (eds.) *Proc. of the IJCAI-91 Workshop on Theoretical and Practical Design of Rational Agents*, Sydney, Australia, August, 1991. Also appears in J. Galliers (ed.), Technical Report 230, Computer Laboratory, University of Cambridge, UK, August, 1991.

I.A. Ferguson. Touring Machines: Rational Planners in Open Worlds. *Proc. of the First Belief Representation and Agent Architecture Workshop*, Cambridge, UK, March, 1990. Also appears in J. Galliers (ed.), Technical Report 194, Computer Laboratory, University of Cambridge, UK, May, 1990.

Acknowledgements

Many thanks to my supervisor William Clocksin for invaluable guidance during the early days of my research and for continued assistance throughout my stay at the Computer Laboratory. I would also like to give a special thanks to Julia Galliers; without her continued encouragement and interest in my work, life in the Lab would have been much less fruitful, interesting, and enjoyable.

I also want to thank various colleagues and friends for their suggestions, comments, and criticisms, many of which went to change — and I believe improve — my ideas and outlook on my work. Here I include Jane Dunlop, Colin Ferguson, Julia Galliers, Raj Goré, Feng Huang, George Kiss, Barney Pell, Han Reichgelt, Arvindra Sehmi, and Thomas Vogel. I am grateful to Steve Pulman for allowing me to attend the Motorway Vehicle Rationality group discussions at SRI Cambridge, and to the University Computing Service for the use of CUS. Kish Shen was very helpful in matters concerning SICStus Prolog, as were Mats Carlsson and Ted Kim regarding XWIP. Martyn Johnson, Graham Titmus, and Chris Hadley provided much needed systems support, and Lewis Tiffany and Paola Bishop were of great assistance during the many hours I spent in the library. I suspect that all of the extra cups of coffee from Edie, Sheila, and Cathy helped in some way too.

My warmest thanks go to my wife, Jane Dunlop, who I feel had to put up with so much, not to mention being dragged several thousand kilometres across the Atlantic so that I could study at Cambridge. I would also like to acknowledge the continued support over the years from Sheena Ferguson, Carson and Nancy Ferguson, Chris Fagan and Siobhan Ferguson, Jerôme Martinez, Bill Westwood, Dick Peacocke, and Moira Norrie.

Without a rather large amount of financial assistance, clearly none of this would have been possible. I would therefore like to thank Bell-Northern Research Ltd., the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom, Clare Hall, IJCAI Inc., the Computer Laboratory, the Cambridge Philosophical Society, and, most definitely, my wife Jane.

Madre, no sé si querrás leerlo, pero esto es para tí...

Contents

1

Introduction

> *Avernus moved out into the middle of the furious activity, machines and robots and train cars rushing quickly all around them. As Avernus stepped into the path of on-rushing vehicles, Derec froze, wanting to pull back. But the expected accidents never took place, the robots and their machines gauging all the actions around them and reacting perfectly to them. That's when the concept of deliberation became clear to Derec. Movement needed to be deliberate, with constant forward momentum. All judgement was based on the idea that movement would be steady and could be avoided once gauged. It was the erratic movement that was dangerous — the abrupt stop, the jump back; down here, such movements would be fatal. Once he understood the concept, it became easier to walk into the path of on-rushing vehicles.*
>
> Mike McQuay, *Isaac Asimov's Robot City*

This dissertation is concerned with the provision of an integrated control architecture for autonomous agents and an associated simulation testbed aimed at facilitating the specification and analysis of agents in a wide range of complex multi-agent task scenarios. The intention is to provide an implementation of a software control architecture which is competent, functionally rich, behaviourally diverse, and which encourages and readily facilitates extensive experimental evaluation. This desire is motivated by the observation that large-scale, computer-controlled systems and facilities are becoming increasingly decentralised and thus potentially stand to benefit from the incorporation of distributed, task-achieving, autonomous agent processes which are more intelligent, robust, and flexible than the predominantly centralised computational processes used at present.

The type of agent process of concern here includes autonomous *mobile* agents which are expected to operate in a dynamic, real-time domain (for example, an automated factory floor or traffic environment) and which are required to carry out reasonably complex, time-constrained tasks in the presence of other similarly goal-directed agents. It is the dynamic, real-time nature of the chosen task-domain which poses most problems for the agents. In particular, the agents — which are assumed to have limited internal computational resources, limited knowledge of other agents' intended tasks, and limited means of acquiring information either from their environment or from other agents — will be required to respond quickly to a number of immediate external threats while at the same time dealing with a host of equally unexpected intra- and inter-agent goal conflicts.

The dissertation presents a new integrated control architecture — the *TouringMachine* agent architecture — which has been designed to provide autonomous, resource-bounded agents with the diverse range of reactive, goal-oriented, reflective, and predictive behaviours typically required in order to carry out complex, time-constrained navigational tasks in partially-structured, dynamic, real-time, multi-agent environments.[1] To analyse the performance of TouringMachines and to facilitate the study of their behavioural ecology — the relationship between each agent's internal configuration, its task environment, and its behavioural repertoire — the architecture has been designed and implemented in conjunction with a feature-rich, instrumented simulation testbed.

1.1 Intelligent Agent Design

The computer-controlled operating environments at such facilities as automated factories, nuclear power plants, telecommunications centres, and space stations are continually becoming more complex. As this complexity grows, it will be increasingly difficult to control such environments with centralised management and scheduling policies that are both robust in the face of unexpected events and flexible at dealing with operational and environmental changes that might occur over time. One solution to this problem which has growing appeal is to distribute — along such dimensions as space and function — the control and scheduling of operations to a number of intelligent, task-achieving *computational* or *robotic agents*. In this dissertation, an *agent* shall be considered to be any goal-directed computational process capable of robust and flexible interaction with its environment.

[1] Henceforth, to avoid further adjectival overload, the partially-structured, dynamic, real-time, multi-agent environments of TouringMachines will often just be described as “complex”.

Most of today's computational or robotic agents are limited to performing a relatively small range of well-defined, pre-programmed, or human-assisted tasks. In order to survive and thrive in complex domains, future agents will need to be made considerably more robust, flexible, and skilled at dealing with exceptional events than they are at present. In such domains agents will typically perform complex tasks requiring some degree of attention to be paid to environmental change, temporal deadlines, computational resource bounds, and the impact the agents' shorter term actions might be having on their longer term goals. On the other hand, time will not stop or slow down for them to deliberate upon all possible courses of action for every world state. Intelligent, resource-bounded agents will thus require a range of skills to monitor and respond promptly to unexpected events, while simultaneously being able to carry out pre-programmed tasks and resolve and recover from unexpected conflicts in a timely and efficient manner.

Real-world domains are likely to be populated by multiple agents, each pursuing any number of tasks. Because agents are likely to have incomplete knowledge about the world and will compete for limited and shared resources, it is inevitable that, over time, some of their goals will conflict. Attempts to construct complex, large-scale systems in which all envisaged conflicts are foreseen and catered for in advance are likely to be too expensive, too complex, or perhaps even impossible to undertake given the effort and uncertainty that would be involved in accounting for all of one's possible future equipment, design, management, and operational changes [Lee89, pages 7 and 182–189].

There is clearly an ongoing *evolution* of the intelligent functions that are present in autonomous control systems. As Antsaklis *et al.* [APW90] suggest, although there are characteristics which separate intelligent from non-intelligent systems, the distinction tends to become less clear as intelligent systems evolve. Thus, systems which might once have been considered intelligent — for example, James Watt's governor for steam engines (see next chapter) — subsequently evolve to gain more character of what are considered to be non-intelligent or numeric-algorithmic systems. Control theory and control systems engineering [DW91, pages 103–175], for instance, provide various numeric-algorithmic methods (for example, finite difference equations and differential equations) for analysing and synthesising control systems for a wide variety of applications. However, for some applications — such as those in which the control systems are constrained to operate with substantial uncertainty (including execution, environmental, and epistemic uncertainty) or those in which the systems must reason about the temporal and causal structure of their environments — access to a richer, more powerful set of representation and reasoning methods might prove more useful; Artificial

Intelligence (AI) provides such methods.[2]

To date, much of the emphasis in AI research has been placed on studying isolated functions and capabilities such as perception, planning, natural language understanding, belief modelling, or learning. In general, little or no attention has been paid to the problem of how to integrate and deploy these capabilities in autonomous machines or systems capable of operating effectively in complex environments. Motivated by the belief that research into integrated agent architectures will serve as one of the main platforms for the development of tomorrow's intelligent robots, many researchers in AI have recently become interested in the challenges of building integrated AI systems which can interact with their surroundings and operate robustly and flexibly in the presence of real-time environmental change and uncertainty.

This dissertation is concerned with the design and implementation of an AI software architecture suitable for controlling and coordinating the actions of a *rational*, *autonomous*, resource-bounded agent embedded in a partially-structured, dynamic, multi-agent world.[3] The research presented in this dissertation involved three complementary efforts: *(i)* understanding the functional and behavioural requirements of intelligent, rational, autonomous, mobile agents for a particular class of real-time, multi-agent domain; *(ii)* realising a particular design and implementation of an integrated agent architecture satisfying the requirements identified; and *(iii)* designing and implementing a highly instrumented and parametrized multi-agent simulation testbed with which to observe and analyse various aspects of agent-level problem solving, coordination, and behavioural ecology.

Operating in the real world means having to deal with multiple events at several levels of granularity — both in time and space. So, while agents must remain reactive in order, say, to survive, some amount of strategic or predictive decision making will be required if agents are to coordinate their

[2] AI methods are a *research* vehicle which help in understanding complex problems and thereby help in organising and synthesising new approaches to problem solving. It should come as no surprise, then, that as solutions to control problems develop, purely algorithmic approaches with more desirable implementation characteristics — such as those used in control systems engineering — will eventually substitute those developed using AI techniques.

[3] The definition of *rational* behaviour used here is borrowed from Bratman *et al.* [BIP88, page 349] and corresponds to "the production of actions that further the goals of an agent, based upon [its] conception of the world." An entity will be considered *autonomous* if — as suggested by Covrigaru and Lindsay [CL91, page 111] — "it is perceived to have goals ... and is able to select among a variety of goals that it is attempting to achieve.", and if, in addition, the entity includes "a measure of complexity; interaction; movement (preferably fluid); a variety of behaviours; robustness and differential responsiveness to a variety of environmental conditions; selective attention; and independent existence without detailed, knowledgeable intervention." [CL91, page 113].

actions with other agents and handle complex, time-constrained goals while keeping their long-term options open. Agents, however, cannot be expected to model their surroundings in every detail as there will simply be too many events to consider, a large number of which will be of little or no relevance anyway. What is required, in effect, is an architecture that can cope with uncertainty, react to unforeseen events, and recover dynamically from poor decisions. All of this, of course, on top of accomplishing whatever tasks it was originally assigned to do. Not surprisingly, it is becoming widely accepted that neither purely *deliberative* nor purely *non-deliberative* control techniques are capable of producing the range of effective, robust, and flexible behaviours desired of future intelligent agents.[4]

The thesis of this dissertation is that it is both desirable and feasible to combine non-deliberative and *suitably designed and integrated* deliberative control functions in a single architecture in order to obtain effective, robust, and flexible behaviours from rational, autonomous, resource-bounded agents which are to carry out tasks in a class of challenging domains — those which are partially-structured, dynamic, real-time, and inhabited by one or more other intentional agents without central authority. A secondary hypothesis which is investigated in this dissertation is the claim that establishing an appropriate balance between reasoning and acting depends heavily on characteristics of the task environments in which the agents are intended to operate.

To address one of the fundamental trade-offs involved in agent architecture design, namely that which exists between an agent's representational power (in other words, the generality and flexibility of its behaviour, the ease with which the designer can program, test, and debug it) and its run-time efficiency, the resulting TouringMachine architecture has been designed by integrating a collection of both deliberative and non-deliberative agent control capabilities. These capabilities include situated action, focus of attention, planning, and various forms of commonsense reasoning about agents' *mental states* — namely, their beliefs, desires and intentions. The architecture is centred on a number of modular, independent, task-achieving control layers which are aimed at ensuring a high degree of operational concurrency. Also, because of the real-time constraints imposed on agents embedded in dynamic environments, an account is taken of TouringMachines' limited computational resources in order to guarantee an upper bound on the latency on its opera-

[4]The term *deliberative* implies that the agent possesses reasonably explicit representations of its own beliefs, plans, and/or goals that it uses in deciding which action it should select at a given time. Conversely, *non-deliberative* implies that the agent's beliefs, plans, and goals are implicitly embedded or pre-compiled into the agent's structure by its designer. Examples of agent architectures which can be classified as deliberative, non-deliberative, and *hybrid* (combining aspects of each approach) will be given in the next chapter.

tions such as sensing, attention focussing, planning, and world modelling. The result is an architecture which can produce a number of reactive, goal-directed, reflective, and predictive behaviours — as and when dictated by the agent's mental state and environmental context.

The research presented in this dissertation adopts a fairly pragmatic approach toward understanding how complex environments might constrain the design of agents, and, conversely, how different navigational task constraints and functional capabilities within the agents might combine to produce different behaviours. To evaluate TouringMachines , a highly instrumented, parametrized, multi-agent simulation testbed has been implemented in conjunction with the TouringMachine control architecture. By enabling the user to specify, observe, and analyse any number of user-customised agents in a variety of single- and multi-agent settings, the testbed provides a powerful platform for the empirical study of TouringMachine behaviour.

It is worth pointing out at this stage that the focus of this research is on the functional and behavioural requirements of *individual* agents which are to carry out *non-shared* tasks in *heterarchical* multi-agent environments. In particular, this dissertation is not concerned with the requirements of agents which are to operate within the framework of a hierarchical organisation or which are to work on joint or collective tasks. In addition to this restriction, the architecture presented in this dissertation ignores the "obvious" requirement that agents be capable of adapting to new environments by learning from their past experiences. It is clear that both skills — that is, being able to interact and coordinate within a social context and being able to improve task performance through some form of machine learning — are both highly desirable (if not essential) for intelligent, autonomous agents operating in multi-agent domains. These concerns lie outside the scope of this dissertation but remain as candidates for future work (see Chapter 9).

1.2 Outline of the Dissertation

Chapter 2 gives a brief overview of some past work on intelligently controlled systems from such fields as engineering, cybernetics, artificial intelligence, robotics, and distributed artificial intelligence. A comprehensive review of several recent examples of integrated agent architectures is then presented, together with a discussion of some of the major requirements, challenges, and desiderata vis-à-vis the design and construction of intelligent autonomous agents.

The TouringMachine agent control architecture is introduced in Chapter 3. An overview is given of the architecture's capabilities and structure, including its three concurrent task-achieving layers, enveloping mediatory control framework, and perception and action subsystem interfaces. TouringMachines' resource-boundedness and real-time operational characteristics are also addressed.

Chapters 4, 5, and 6 expand on the major design, operational, and implementational issues relating to the three control layers which together constitute the core of the TouringMachine architecture: the $\mathcal{R}$ (reactive), $\mathcal{P}$ (planning), and $\mathcal{M}$ (modelling) control layers, respectively.

Chapter 7 describes the TouringWorld Testbed: the instrumented multi-agent simulation testbed which enables controlled empirical investigations on the dynamic behaviour of user-customised TouringMachine agents under a range of user-specified environmental conditions. This chapter also provides a detailed characterisation of the particular multi-agent navigation domain which has been chosen as the platform for implementing and evaluating the TouringMachine architecture.

Some experimentation with the TouringMachine agent architecture and TouringWorld Testbed is described in Chapter 8. Various aspects of TouringMachine performance and behavioural ecology are studied and discussed through a number of single- and multi-agent scenarios. Issues and ideas for further evaluation are also presented.

Chapter 9 provides a summary of the dissertation and major contributions of the research. Several limitations about and ideas for enhancing the present agent architecture are given; finally, some concluding remarks are made about the dissertation.

Appendix A gives the complete extended BNF grammar defining the Testbed's user-level language for parametrizing TouringMachine agents and TouringWorld scenarios and environments.

2

Intelligent Agent Design

... the law ought always to trust people with the care of their own interest, as in their local situations they must generally be able to judge better of it than the legislator can do.

Adam Smith, *An Inquiry into the Nature and Causes of the Wealth of Nations*

2.1 History and Evolution

The quest to design intelligent control in artificial systems dates back a long time. Well before electronic computers had been invented, for instance, the engineer James Watt (1736 – 1819) popularised the use of mechanical *feedback control* as a way of automatically regulating the velocity of rotation in steam engines, thereby controlling their energy intake. These feedback control devices, known as *governors*, were designed to refine actions and produce stability in dynamical systems via a process called *negative feedback*; in other words, by feeding output from the system back to its input as a means of comparing actual and intended performance so that compensatory changes in the input could then be undertaken [Gre87, pages 259–261].

Almost two centuries later, the concept of feedback would also prove fundamental in the development of the field of *cybernetics*. A highly cross-disciplinary field, cybernetics[1] was launched in the forties and fifties by the likes of Norbert Weiner, Ross Ashby, and Grey Walter, their principal aim being to unify mathematically the disparate studies of control and communication in animals and machines. Much of the inspiration behind their work

[1]The term *cybernetics* derives from the Greek *kubernetes* meaning “governor”.

came from the deployment during World War II of *analog* electronic computers — machines which simulated physical systems by representing their changing quantities as analogous moves of shafts or voltages — for such tasks as navigating aircraft, controlling anti-aircraft guns, and precision target bombing. Some of the developers of these early analog computers were struck by the apparent similarity between the internal operations of the computers and the regulatory systems in living beings (for example, the homeostatic control of body temperature and blood glucose levels within humans) and so became inspired to build machines which could learn and which would act as though they were "alive" [Mor88, pages 6–17].

Due to a combination of representational and technological difficulties, the field of cybernetics would thrive for less than two decades. With the arrival of substantially more powerful digital computers — and thus the ability to perform problem solving of unprecedented complexity — another field, *Artificial Intelligence* (AI), would continue the pursuit toward developing intelligent artificial systems. Inspired by the pioneering research of such people as John von Neumann, Alan Turing, and Claude Shannon (all of whom cherished the hope that the ability to think rationally might one day be performed by a machine) much of the early work in AI would adopt the view of problem solving as formal theorem proving or as a heuristic search over a space of goal-subgoal structures.

Since the foundational work of such researchers as Alan Newell, Herbert Simon, John McCarthy, and Marvin Minsky, the field of AI has seen the appearance of a host of artificial systems specialised — more or less successfully — in game playing, mathematical discovery, diagnosis, planning, and natural language understanding, among others. Unlike the sensorimotor-level and biologically-inspired artifacts produced by the cyberneticists, AI researchers have by and large tended to concentrate on building narrowly defined competence-oriented systems based on the formalisation and simulation of human intellectual reasoning processes.[2]

In 1950 Alan Turing suggested there might be two ways toward the goal of creating an intelligent machine. The first way — which Brooks calls the *unembodied* path [Bro91a, page 573] — would be to concentrate on programming intellectual activities such as playing chess; the second — the *embodied*

[2] This characterisation of AI research, admittedly, only describes systems of the class identified by the *physical symbol system hypothesis*: namely, the class of systems which embodies the "essential nature of symbols" and which is "the necessary and sufficient condition for a generally intelligent agent" [New82, page 94]. Non-symbolic approaches to AI — for example, those which are embodied in connectionist or artificial neural networks and in genetic algorithms — have been omitted from the present discussion but can be read about elsewhere [RK91, pages 483–528].

path — would be "to provide the machine with the best sense organs that money can buy, and then teach it to understand and speak English." [Tur50, page 460].[3] While work in AI has almost exclusively followed the first path, a relative of AI, *robotics*, would emerge in the mid-sixties and follow the second of these paths.[4] Influenced by Minsky's seminal work at the Massachusetts Institute of Technology involving the coupling of sophisticated computer control programs to television cameras and mechanical robot arms, a number of *mobile* robotics projects would soon begin to emerge at other institutions; most notably the Shakey project led by Charles Rosen and Nils Nilsson at the Stanford Research Institute, and the Cart project of John McCarthy, Les Earnest, and Hans Moravec at the Stanford Artificial Intelligence Laboratory [Mor88, pages 14–20].

While many industrial robots have recently found their way into a number of reasonably well-defined (and typically non-mobile) niches, progress in robotics toward creating highly versatile, programmable, intelligent machines which can systematically extract and manipulate information from their environments and carry out complex tasks has been no faster, it would appear, than that achieved to date in AI.

The construction of intelligent artificial systems, or *agents* as they are now commonly called, is also the goal of researchers in *Distributed Artificial Intelligence* (DAI) — the branch of AI concerned with concurrency and distribution in AI computations. This is most evident, perhaps, in the subfield of DAI known as *Multi-Agent* (MA) systems which deals with the coordination of intelligent behaviour among a collection of autonomous, intelligent agents.[5] While an advance on much of the earlier work in AI (in the sense that it deals with agent-level coordination and problem solving among *multiple* agents) much of the initial work in DAI was restricted to considering strictly benevolent [Len75], fully knowledgeable [RG85] and/or resource-*un*bounded agents working on very well defined shared tasks [Geo83] or operating in highly

[3] Wilson [Wil90] also refers to this approach as the *animat* (or artificial animal) approach: one which is "holistic, focussing on complete systems ... that, like animals, exist in realistic environments and must cope with the varied problems that they present."

[4] The term *robotics*, in fact, first appeared in the 1942 science fiction story 'Runaround' by Isaac Asimov [Asi50]. The term itself derives from the Czech word for *worker* and would first appear in the English language in 1923 via a translation of Karel Čapek's play 'R.U.R' (Rossum's Universal Robots). The widespread popularity of the play helped replace the term in vogue at the time — *automaton* — for the now more commonly used term *robot*.

[5] For the interested reader, an excellent overview of the major problem areas in DAI (for example, task description and decomposition, inter-agent communication, coordination and global coherence, agent modelling, resolving inter-agent disparities), as well as some of the historical antecedents of and current research in DAI can be found in Bond and Gasser [BG88, pages 3–35].

structured task environments in which inter-agent conflict had been all but programmed away [TMC81]. More recently though, researchers in DAI have begun to consider more realistic, less well structured domains comprising groups of decentralised autonomous agents which are constrained by limited resources and capabilities, which have *bounded rationality*,[6] and which, by virtue of the inherent complexity and unpredictability in their environments, can at best be expected to behave *satisficingly* with respect to their goals: that is, by using decision methods which produce "good enough" rather than optimal behaviour [Sim81, page 35].

Most recently, a number of researchers from the fields of AI, DAI, and robotics have started to turn their attention to the problem of designing and implementing integrated control architectures for intelligent agents, or, as they are also known, *integrated agent architectures*. An integrated agent architecture, Drummond and Kaelbling [DK90] suggest, is a theory or paradigm by which one may design and program intelligent agents. Typically targeted for use in dynamic, unpredictable, and often multi-agent environments, an intelligent agent can be regarded as a structured collection of sensors, computers, and effectors; in this structure, the sensors measure conditions in the world, the computers process the sensory information, and the effectors take action in the world. Since changes in the world realised by the agent's effectors will close the loop to the agent's sensors, the agent can be described as being *embedded* in its environment.

A number of different integrated architectures have been proposed recently, each one aimed at providing agents with a particular level of intelligent, autonomous control. Broadly speaking, the different approaches can be classified according to the mechanism for action selection which the agent uses when determining what to do next. In particular, if the agent selects actions by explicitly deliberating upon the various options that are present (for example, with the use of an internal symbolic world model, via a search of its plan space, or by considering the expected utility of available execution methods) the agent can be considered *deliberative*.[7] Alternatively, if the agent's choice of action is situationally determined — in other words, pre-programmed or in some way "hardwired" to execute given the occurrence of a particular set of environmental conditions — then they can be described as *non-deliberative*.[8]

[6] An agent has *bounded rationality* if it is required to operate in "situations where the complexity of the environment is immensely greater than the computational powers of the [agent]." [Sim81, page 190].

[7] Genesereth and Nilsson prefer to describe such agents as "deliberate" [GN87, pages 325–327].

[8] The distinction between deliberative and non-deliberative agents is similar to the distinction made by Simon [Sim81, pages 31–34] between systems exhibiting *procedural rationality*,

Also of interest are architectures in which the choice of action is realised by using some combination of deliberative and non-deliberative techniques. Such agent architectures can be called *hybrid*.

The subject of this dissertation falls within the area of integrated agent architecture design. As one might expect, integrated agent architecture research shares roots with many of the research fields mentioned above. For example, the architecture presented in this dissertation — the TouringMachine agent control architecture — can be seen to borrow techniques from control systems engineering (for example, self-regulation through negative feedback), AI (for example, planning, causal reasoning), robotics (for example, mobile domain, collision detection and avoidance), and also DAI (for example, task coordination through modelling of agents' actions and plans, recognising and reconciling conflicting intentions among a collection of agents).

The design principles behind any control architecture must appropriately reflect certain requirements of the different tasks and environments for which it is being developed. This is true also of the TouringMachine control architecture. However, before discussing the particular criteria which were used in arriving at the TouringMachine architecture, it would be useful — primarily to characterise the various strengths and weaknesses of the different extant design approaches — to review some representative examples of deliberative, non-deliberative, and hybrid agent architectures. Given the fairly recent surge of interest in integrated agent architecture research and consequent proliferation of proposed designs, it would be quite difficult to review all existing agent architectures in this dissertation. As a result, the following three sections will be limited by focussing on architectures which are relatively mature and which, arguably perhaps, have been the most influential or have made the largest impact in the field.[9]

2.2 Deliberative Architectures

Deliberative agent architectures have their roots in the *sense-plan-act* problem solving paradigm of classical AI planning systems such as STRIPS [FN90] and NOAH [Sac90]. The aim of these planning systems was to generate provably correct action sequences or *plans*, which, upon separate execution, would have the effect of achieving some desired goal state. The success of such planners

---

that is, systems which *compute* the rational thing to do; and systems exhibiting *substantive rationality* which simply *do* the rational thing.

[9]For the interested reader, ACM SIGART Bulletin, Vol. 2, No. 4, 1991 [Lai91] includes a special section on Integrated Cognitive Architectures containing discussion papers on a number of architectures not reviewed here.

rested on a number of important assumptions; namely, that the system would have complete and up-to-date knowledge about the state of its world, that the effects of its actions (as represented in its plan *operators*) would always be correct and known in advance, and that the state of the world would remain unchanged unless explicitly acted upon by the planner itself.

To deal with the uncertainty of generating plans in more complex dynamic worlds, a number of planning systems have since been developed which can interleave plan formation and execution (for example, NASL [McD90]) and which can also monitor the execution of planned actions and initiate some form of plan modification or *replanning* should the actions not achieve their desired effect (for example, AUTOPILOT [TMC81] and ELMER [MRS82], and the domain-independent planners SIPE [Wil85] and IPEM [AIS90]). The plans generated by these systems typically contain explicit descriptions of the conditions that are required to hold for correct plan execution; these conditions are then periodically checked by the planner at execution time.

Planners which can interleave generation and execution are typically less susceptible to errors caused by changes in the environment and thus more able to cope with a degree of execution uncertainty. However, the planning techniques typically employed by these systems can be very time-consuming and so are not likely to be suited to domains where replanning is frequently necessary, where the planner's initial goals and intentions may themselves need modifying at some stage, or where the time available for planning an action varies depending on the planner's situational circumstances [Geo90].[10] More recently, however, a number of sophisticated planning systems have been developed which, to a greater or lesser degree, are able to meet some of these requirements. Some of these are now described.

2.2.1 IRMA — Bratman et al.

The *Intelligent Resource-Bounded Machine Architecture* (IRMA) [BIP88] is a *practical reasoning system* — a system by which an agent forms plans — aimed at producing rational behaviour in resource-bounded agents by including a number of mechanisms which limit the amount of computation to be performed. IRMA is a *Belief/ Desire/ Intention* (BDI) architecture as it includes fairly direct representations of the agent's beliefs, desires, and intentions. Agents' intentions are viewed as being structured into larger plans, and, besides

[10] As Bratman [BIP88] notes, replanning modules in such planners typically have the same drawbacks as the planning modules themselves: as well as operating under the assumption that the world around them is fixed during execution, they usually require complete and correct information about all possible unexpected events.

their functional role in producing actions, plans in IRMA are also viewed as having a role in constraining the agent's options for further reasoning.

The architecture incorporates a number of modules including an intention structure (a time-ordered set of partial, tree-structured plans), a means-end reasoner, an opportunity analyser, a filtering process, and a deliberation process. The means-end reasoner suggests options for action by considering the agent's partial plans. Further options are suggested by the opportunity analyser which monitors the agent's environment for pertinent dynamic events. All options are passed to the filtering process where they are tested — using a *compatibility filter* — for consistency with the agent's existing plans. Surviving options are then passed to the deliberation process which, after due consideration, will decide which new intention to adopt.

In addition to a compatibility filter, the filtering process contains a *filter override mechanism* — effectively a user-level control knob which determines the agent's environmental sensitivity by controlling the number of new options the agent should consider for deliberation. Single-agent experiments with the IRMA architecture using a number of different meta-level reasoning (deliberation and filtering) strategies under a range of different environmental conditions have been studied using the Tileworld simulation testbed [PR90].

2.2.2 AUTODRIVE — Wood

Motivated by the desire to address the different human task requirements involved in vehicle driving, the AUTODRIVE architecture [Woo90] enables agents to plan routes in a simulated multi-agent traffic environment. The architecture is centred around a planner which integrates traditional problem solving (the generation of hierarchical, temporally ordered route plans) with a process called *dynamic goal creation*: the continual run-time creation and modification of planning subgoals which, while aiming to address the planner's changing situational constraints, simultaneously strives to bring the agent nearer its goal.

This style of plan generation is based on the notion of isolating the planner's high-level goal stability (for example, its fixed, unchanging route plan) from its subgoal or lower level action instability (for example, whether or not it will encounter and have to stop at a red light). In order to identify potentially relevant subgoals or situational constraints for incorporating into the plan at run-time, the agent's planner is extended with various modules for performing *dynamic world modelling*: namely, perception, heuristic focus of attention, and plan/intention recognition. By recognising other agents' plans, AUTODRIVE agents are able to associate observationally-derived information with that which can only be inferred abstractly. This then enables them to make dynamic changes to their plans based on their predictions of possible future conflicts or interactions.

2.2.3 Behaviour Hierarchies — Durfee and Montgomery

Durfee and Montgomery [DM90] have experimented with a number of blackboard-based agents which autonomously and flexibly coordinate their own activities by exchanging information about their intended task-related behaviours. The agents, which set out with no prior knowledge about each others' tasks, employ a hierarchical negotiation protocol to exchange — iteratively at differing levels of abstraction — information about their anticipated behaviours. The protocol is centred on the notion of a *behaviour hierarchy* which is used by agents to represent their own behaviours and which, upon comparison with the behaviour descriptions received from other agents, can be used by the agent to discover potential task interactions.

Behaviour hierarchies subsume the traditional notions of plan and goal hierarchies and can be used to represent several composition/decomposition dimensions of an agent's behaviour; namely, *who* the agent is, *what* it is trying to achieve, *when* it is trying to achieve its task, *where* it intends to carry out its task, *how* it is going to carry out its task, as well as *why* it is doing what it is doing. Agents map received behaviours into their 6-dimensional local behaviour representations and then proceed to identify interacting behaviours which might have arisen through particular resource conflicts. Agents, which are given unique authority levels to resolve such conflicts, can then decide whether to continue by using an alternate, non-interacting behaviour (for example, one that differs in terms of space or time), or whether to exchange increasingly more detailed information in order to identify, in more specific terms, how or whether the behaviours actually interact. Scenarios involving up to three mobile agents, each attempting to plan collision-free paths through a common area with shared access points have been analysed using the MICE testbed [DM89].

2.2.4 Agent-Oriented Programming — Shoham

Shoham [Sho90] has proposed a computational framework called *Agent-Oriented Programming* (AOP) for the specification and programming of artificial agents. Agents are defined as entities possessing formal versions of mental state; that is, formal versions of knowledge, belief, and the like. In their current form, AOP agents can be considered purely deliberative. In addition, AOP agents are assumed to be operating in non-physical environments without any time pressure.

From an engineering point of view, AOP can be regarded as a specialisation of object-oriented programming in which the objects — in this case, agents — can possess various *mental states* about themselves and about one another. From a formal point of view, AOP can be regarded as a specialisation of a formal language (a temporalised epistemic logic of belief) augmented with various modal operators such as *commitment*, *choice* (commitment to oneself), and *capability*.

The AOP framework includes three components: *(i)* a restricted formal language for describing an agent's mental state — specifically, an explicit-time, point-based temporal logic with the two basic modalities of belief (the standard KD45 operator `B`) and commitment (a ternary KD4 operator `CMT(a,b,`ϕ`)` signifying that the agent `a` is committed to `b` about ϕ); *(ii)* an as yet unimplemented interpreted programming language whose semantics are to be derived directly from the semantics of agent mental state; and *(iii)* a compiler based on Rosenschein and Kaelbling's compiler for *situated automata* (see Section 2.3 below) from the agent-level language to an abstract model of processes.

2.2.5 Homer — Vere and Bickmore

Homer [VB90] is an integrated AI artifact which is embedded in a simulated, single-agent, object-cluttered marine environment called the Seaworld. Homer integrates a number of deliberative capabilities including limited natural language generation and recognition (using an 800-word English vocabulary and medium coverage grammar), temporal planning and reasoning, acting on and perceiving its environment, as well as the ability to reflect upon its own experiences.

Homer's operations are centred around a temporal planner which it uses to synthesise plans in response to a human user's natural language goal commands. Goal commands typically include time constraints on their achievement and preservation. The planner then imposes goal protection conditions which are constantly monitored in order to detect plan violations, both at plan time or later during execution. Associated with the planner is a set of declarative activity models in precondition/postcondition format, describing all of the actions, inferences, and events that the agent knows about. Homer employs one of these actions — the "go" action — to plan collision-free trajectories to desired locations. Besides being able to form and retain compound future plans, Homer's planner is also capable of limited replanning in order to accommodate additional goals that are given to it.

Homer sets out with limited knowledge of its world, eventually gaining new information about the different objects in its world either by perceiving them or by being told about them directly by the user. Homer time-stamps and records all events which have transpired in its life by placing these in its *episodic memory*. Homer's events include its perceptions, all of the goal commands and questions it has previously received from the user, as well as any actions that it has taken. With the use of its temporal planner and various reflective processes for monitoring and processing its personal memory, Homer is able to make inferences and provide answers to a range of questions about its past experiences, present activities and perceptions, as well as its future intentions.

2.3 Non-deliberative Architectures

A number of new architectures have recently been proposed which, when compared to the deliberative architectures described above, adopt a radically different stance vis-à-vis how they select which actions to take. Non-deliberative architectures — including those often referred to as *reactive*, *situated*, or *behaviour-based* — typically make all necessary control decisions at run-time on the basis of limited amounts of information (usually only that which is currently available from their sensors), limited internal state, and with a minimal amount of inference [HF90].[11]

Whereas traditional planners are required to produce optimal or *correct* actions, non-deliberative architectures are designed to produce *robust* actions. Typically built from relatively simple control mechanisms — for example, finite-state machines (FSMs) or domain-specific stimulus-response rules — the design of such architectures is motivated, in part, by Simon's hypothesis that complex behaviour in an agent need not necessarily be the product of a complex internal design; rather, the complexity of the agent's behaviour may simply be a reflection of the complexity of the environment in which it operates [Sim81, page 64].

The common emphasis in these systems is put on, among other things, fairly direct coupling of perception to action, decentralisation of control, dynamic interaction with the environment (characteristics of the environment are exploited to serve the functioning of the agent), as well as intrinsic mecha-

[11] In the extreme case where the agent has no internal state (its activity, therefore, being determined *entirely* by its current environment), it is referred to as a *tropistic* agent [GN87, pages 307–311]. Most, if not all, existing non-deliberative architectures do make use of some internal state; what all of them avoid, however, is the need to store and maintain complete and correct internal models of their worlds.

nisms for dealing with innate resource limitations and incomplete knowledge [Mae90]. To gain a better understanding of how such systems work, a number of non-deliberative agent architectures are described below.

2.3.1 Subsumption Architecture — Brooks

Brooks has developed a methodology and architecture — the *subsumption architecture* — for controlling a number of mobile robots known as *artificial beings* [Bro86] or *Creatures* [Bro91b]. The subsumption architecture is a layered controlled system in which each layer, rather than implementing some individuated control function such as perception, memory, attention, or judgement, realises instead a particular task-achieving behaviour or domain-specific *competence* (for example, obstacle avoidance, random wandering). This way of dividing up an agent's faculties is often referred to as a *vertical* decomposition, in contrast with the more conventional *horizontal* method of faculty decomposition [Fod83, pages 10–23] used by, among other things, traditional planning systems.

Each layer in the subsumption architecture is composed of a fixed-topology network of FSMs, together with one or more data registers and internal timing units. Layers communicate with each other using fixed-length messages over low-bandwidth channels or *wires*. The FSMs within a layer can be made to change state upon the arrival of messages from other layers or after the expiration of designated time periods. Layers operate asynchronously and in parallel with each other and do not employ any shared global memory.

Control is layered (non-hierarchically) with higher level layers subsuming the roles of the lower level layers when they wish to take control. In particular, layers are able, for finite pre-programmed time periods, to substitute (*suppress*) the inputs to and remove (*inhibit*) the outputs from lower level layers, and thus affect the normal flow of data within the layers. Careful programming of this inter-layer control will have the overall effect of "biasing" the agent's actions toward achieving its higher level goals while still attending to its lower level or more critical goals. Robots controlled using the subsumption approach have been successfully deployed for such tasks as indoor room exploration, map building, and simple route planning [Bro91a].

2.3.2 Situated Automata — Rosenschein and Kaelbling

Rosenschein and Kaelbling have developed an architecture for real-time operation based on the foundations of *situated automata theory* [Kae87, Kae91]. Situated automata theory is a formal semantics of embedded computation which gives a specification of the information content of the internal states of a machine in terms of the external states of the environment in which the machine is embedded. This specification can be described thus: when a machine is in state s it can be said to carry the information that φ iff whenever the machine is in state s, the proposition φ holds in the environment; in such a situation, the machine is said to "know" φ.

This theory of computation has lead to a programming methodology for embedded agents in which an agent is viewed as performing a transduction from the stream of perceptual inputs that it receives from the environment to a stream of actions that it then effects upon the environment. Computation within an agent is modelled as a finite-state machine, expressed as a fixed-depth sequential circuit which is guaranteed to perform its computations in constant-time bounded steps.

The situated automata approach is designed to allow the automatic compilation of high-level task/environment descriptions into low-level reactive control mechanisms. To assist in this task, the authors have developed Gapps, a high-level goal-reduction language which facilitates the programming of an agent's action component by automatically generating appropriate combinational logic networks (condition-action rules), and Rex, a Lisp-like hardware description language for generating low-level executable sequential circuits.

2.3.3 Pengi — Agre and Chapman

Pengi [AC87] is an autonomous agent (a penguin) that operates in a simulated world (a real-time video game) which it cohabits with a number of hostile predator agents (killer bees). Pengi's design was inspired by investigations into the dynamics of routine activity, the results of which would suggest, according to Agre and Chapman, that activity mostly derives from very simple sorts of machinery interacting with an agent's immediate situation.

Pengi's activity is based on the notion of a *routine* — a pattern of interaction between the agent and its world. Routines are opportunistic and are typically not represented within the agent itself. An agent engaging in a routine is not driven by any preconceived notion of what will happen; rather, when the situation changes, other responses simply become applicable. Such an agent can be said to *improvise* its actions. In addition, Pengi's activity is also guided by focussing on relevant properties of the immediate environment. Agre and Chapman call such properties the agent's *indexical-functional aspects*. In such a framework, the agent accesses the world by using terms in an agent-centred ontology (for example, *the-block-I'm-pushing*) rather than by using a more traditional and computationally more expensive objective ontology such as first-order logic which would require the instantiation of variables.

The agent architecture is composed of a central system and a peripheral system. The central system is implemented as a combinational network of situation-action rules and is responsible for selecting actions that are appropriate given the agent's present situational circumstances. The peripheral system is responsible for processing perceptions and for effector control. Guided by the central network, and with the use of a set of domain-tailored *visual routines*, the peripheral system is charged with identifying, marking, and indexing those aspects — that is, the intrinsic features and properties of particular local objects — in the environment which are deemed relevant to the agent's goal.

2.3.4 Reactive Action Packages — Firby

Firby has developed a reactive planner based on the notion of *Reactive Action Packages* (RAPs) [Fir87]. RAPs are effectively autonomous processes which pursue a given planning goal until that goal is achieved. If the planner has several goals, then it will correspondingly have several independent RAPs, each trying to achieve its own particular goal.

The reactive planner is composed of a RAP execution queue, a RAP interpreter, a constantly updated current world model, as well as a hardware interface to the robot's sensors and effectors which also provides feedback to the planner's world model about action execution failures and/or successes. RAPs consist of a predefined set of methods for achieving some particular goal. Methods, which are annotated with suitable applicability constraints, are either primitive commands — actions sent to the robot hardware interface — or consist of a *task net* — a partially ordered network of subtasks.

RAPs sit on the planner's execution queue and wait to be selected by the RAP interpreter. RAPs are selected on the basis of the planner's approaching temporal deadlines and according to any ordering constraints which reside in the RAPs' task nets. When selected, a RAP will consult the world model and, based solely on the current situational information that it finds, will either issue a command to the hardware interface or will be interpreted through the processing of its task net. Such processing has the effect of placing a RAP's task net (subgoal) commands on the execution queue for subsequent processing. RAPs will be returned to the execution queue if they are waiting for some subgoal to execute: at that point, another RAP may be selected, so RAP execution effectively becomes interleaved. Since each RAP has control over the execution queue for no more than one cycle at a time, and since inferences made on the current world model are always tightly constrained (for example, no backward inferences are performed on queries and no predictions are made of the agent's future state) the planner can guarantee a high degree of reactivity.

2.3.5 Universal Plans — Schoppers

Universal plans [Sch87] are a representation for agent behaviours that specify appropriate actions (reactions) for all of the situations that can be classified and perceived within a particular domain. A universal plan is effectively a highly conditional linearised plan or decision tree which, given any initial starting state in the environment, can map each possible world state to a specific action to be taken by the agent. Unlike the actions in the plans of traditional planners, actions in a universal plan are selected via a classification of the actual situation encountered at execution time. In order to do this, the universal plan must explicitly identify all of the world state predicates that will need to be monitored at execution time.

Universal plans are compiled in advance with a nonlinear planner using a process of backchaining and goal-reduction on a set of state-space STRIPS-like operator schemas. The decision tree that results from the compilation process embodies the agent's top-most goal condition (root node of the tree) plus the various subgoals (remaining nodes) of this top-most goal. During execution, the compiled plan is interpreted in order to find, at each time instant, the action that is appropriate for the current state of the world. Initially, the agent's top-most goal is achieved by backchaining down through its precondition subgoal arcs, looking for any node (operator) conditions that are false and then identifying whatever actions need to be taken to make the particular node preconditions true. Once an appropriate action is found, it is executed continuously until the truth value of some state predicate changes, after which the tree is traversed again in order to find the next appropriate action.

If the world is cooperative long enough, the planner's actions will have their intended effect and so its behaviour, in the long run, will be goal-directed. If, on the other hand, an action fails to achieve its desired effect, no replanning (in the traditional planning sense) is necessary, but rather only the selection of a new initial point from which to execute the plan.

2.3.6 Dynamic Action Selection — Maes

Maes [Mae90] proposes an action selection algorithm which views an autonomous agent as a collection of *competence modules*. The modules resemble operators in a classical planning system: in other words, they specify the preconditions necessary to become active, as well as their expected effects, the latter being expressed using a STRIPS-like add-/delete-list representation. In addition, each module has an associated *activation level*. Like the situation-action rules in the agent Pengi described above, Maes' modules eschew variables, making use instead of the agent's indexical-functional aspects.

Competence modules are inter-linked in a network using three types of links (successor, predecessor, and conflicter) that indicate which modules make references to each another — this information resides within their precondition lists and/or their execution add- and delete-lists. The modules use these links to activate or inhibit each other so that after some time the activation energy accumulates in those modules which represent the "best" actions to take. Modules are selected for execution when their activation levels reach some pre-defined threshold.

Input of activation energy comes from both the currently observed situation (represented as a set of propositions) and from the global goals of the agent. Inhibitions are carried out by those goals of the agent's which have already been achieved or which need subsequently to be protected. The global behaviour of the action selection algorithm is mediated through a collection of user-level parameters which are used to set such things as the module activation threshold (θ), activation energy levels injected into the network by observed propositions (ϕ) and goals (γ), and the activation energy consumed by protected goals (δ). In Maes' algorithm the control structure which regulates when a particular action gets activated is *emergent*: the dynamics of interaction between the actions (modules) themselves establishes the sequence of selected actions in a completely distributed way.

2.4 Hybrid Architectures

A critical problem facing a non-deliberative agent is that should its environment diverge enough from that for which it was originally designed, then the agent may end up producing inappropriate behaviours. While such an agent might be robust enough to avoid producing harmful or fatal behaviours — perhaps even over extended periods of time — it is not so clear whether it would also be capable of behaving in such a way that it was able to carry out its intended long-term tasks in an effective manner. To address such a problem, a number of architectures have recently been proposed which, to a greater or lesser degree, aim to provide agents with appropriate reactive capabilities, while at the same time permitting high-level, deliberative and predictive reasoning about plans or goals. A number of these hybrid architectures are described below.

2.4.1 Procedural Reasoning System — Georgeff et al.

The *Procedural Reasoning System* (PRS) [GI89, KG91] is a system for controlling and carrying out the high-level reasoning of a robot that combines traditional means-end reasoning with the abilities to react to unanticipated events and to change goals and intentions as situations warrant. PRS has been used to control the movements of a real robot operating in an single-agent, indoor navigation domain [GLS87].

A PRS agent comprises a set of changing *beliefs* (facts about the world), a set of current goals or *desires*, a set of procedural plan schemas or *Knowledge Areas* (KAs) which describe how to achieve its goals and how to react to particular events, an interpreter for manipulating each of these components, and a process stack of currently active KAs or *intentions*. PRS operates as a partial, hierarchical planner, its interpreter permitting the interleaving of planning and execution. As KA interpretation is time-bounded, a guaranteed level of responsiveness can always be maintained.

The set of KAs also includes *metalevel* KAs. Interpreted in the same manner as regular KAs, these metalevel routines contain application-specific knowledge which instruct the agent how to manipulate its own beliefs, desires, and intentions; how to prioritise and select among conflicting KAs; and how to make best use of the available resources given the system's changing real-time constraints. The agent's beliefs and goals determine which KAs are to be considered for execution and, as KAs are executed, new subgoals will be posted and new beliefs derived.

2.4.2 Adaptive Intelligent Systems — Hayes-Roth

An *Adaptive Intelligent System* (AIS) is a knowledge-based system that reasons about and interacts with other dynamic entities in real time [HR88, HR90]. For an AIS to be effective, it must be capable of perception, action, cognition, and attentional focus.

Hayes-Roth has developed an AIS called GUARDIAN for use in an intensive care patient monitoring application. GUARDIAN comprises a cognitive component, a set of asynchronous I/O subsystems, a set of dynamic I/O channels, and a satisficing reasoning cycle. Implemented as a parallel blackboard system, the cognitive component performs general-purpose reasoning by engaging in *dynamic control planning*. This is a process for incrementally constructing and modifying *control plans*: temporally ordered patterns of control decisions, each of which describes a class of operations the agent intends to perform during a particular period of time. Operations within the cognitive component are cyclically processed in turn by an *agenda manager*, an operations *scheduler*, and an operations *executor*.

GUARDIAN's asynchronous I/O subsystems integrate perception with cognition, as well as relaying intended actions to the agent's effectors. The dynamic I/O channels integrate the agent's sensors and effectors with the cognitive component by implementing a range of pre-processing and heuristic selective attention functions. GUARDIAN employs a satisficing reasoning cycle in order to provide the guaranteed latency required for real-time operation. This is achieved through the use of a heuristic operations control policy, the role of which is to limit the total number of operations to be processed by the cognitive component. The policy is defined by a set of *cycle parameters* which, with the use of feedback from the I/O channels, can be fine-tuned at run-time for improved system performance.

2.4.3 Dynamic Reaction — Hendler et al.

Hendler *et al.* have recently developed a collection of agent architecture designs based on the notion of *dynamic reaction*, a set of techniques for managing observation and action in dynamic domains [SH88]. Most recently, these techniques have been extended to interact with a planning system called the *Abstraction-Partitioned Evaluator* (APE) architecture which has been tested in a simulated, single-agent, indoor navigation domain [SH90].

The APE architecture is composed of a number of concurrent, hierarchically abstract action control layers, each representing and reasoning about some particular aspect of the agent's task domain. Implemented as a parallel blackboard-based planner, the five layers — sensor/motor, spatial, temporal, causal, and conventional (general knowledge) — effectively partition the agent's data processing duties along a number of dimensions including temporal granularity, information/resource use, and functional abstraction. Perceptual information flows strictly from the agent sensors (connected to the sensor/motor level) toward the higher levels, while command or goal-achievement information flows strictly downward towards the agent's effectors (also connected to the sensor/motor level).

Besides mechanisms for communicating with other layers, each layer in the APE architecture comprises a number of other modules which include a NASL-like planner which is capable of interleaving plan generation and execution, a collection of NOAH-like plan operators for carrying out various reactive and deliberative tasks, and a *State of Affairs* (SOA) structure which contains an up-to-date model of the agent's world as well as records of the agent's current goals. The plan operators also contain a number of *monitors* — independent information gathering components whose duties include sending reports to the SOA model of any significant observed events or plan constraint violations and initiating replanning and taking action in the light of predicted exceptions.

2.4.4 Phoenix — Cohen et al.

The Phoenix project [CGHH89, HHC90, HC90] is an investigation into real-time agent *behavioural ecology* — that is, the functional relationships between the designs of agents, the environments in which they operate, and their resulting behaviours. Phoenix is also a real-time, adaptive planning architecture used for controlling a variety of different autonomous and semi-autonomous agents embedded in a simulated forest fire domain.

Phoenix agents are composed of two parallel and nearly independent mechanisms for generating actions. The first of these, the *reflexive* component, is designed to generate immediate reactions to a range of different environmental situations. The second mechanism, the *cognitive component*, is charged with performing longer term, computationally expensive planning. Each component is independently connected to the agent's sensors and action effectors.

The cognitive component has ultimate control of the agent's actions. Its main duty is to instantiate and execute stored plans using a method of deferred commitment called *lazy skeletal refinement*. In addition, the cognitive component is also responsible for responding to interrupt flags set by the agent's reflexive component, for handling communications with other agents, and for performing such functions as plan selection and scheduling, error recovery and replanning, as well as monitoring of the agent's plans and activities. Activity monitoring is accomplished with the use of *envelopes* — in-plan, predictive processes which can be used both to assess expectations of progress and also to generate recovery actions or plans when these expectations are not met.

2.4.5 Decision-Theoretic Control — Ogasawara

Ogasawara [Oga91] has developed a mobile robot control system for navigating and map building in a simulated environment containing a number of fixed and moving obstacles. The architecture is composed of a set of task-oriented, subsumption-like *behavioural modules*, with one for each of the three tasks performed by the agent: Avoid Obstacles, Get to Goal, and Build Map. The architecture also comprises a centralised world model structure and an arbitration procedure for mediating and selecting among the agent's competing behaviours.

Each behavioural module has access to a number of problem solving strategies, each differing in terms of its computational cost or solution accuracy. Behavioural modules store local (per-module) decision-theoretic formulations of the utilities of specific outcome states (for example, hitting a wall) and the probabilities of each problem solving strategy yielding a particular action sequence output. Once computed, behavioural modules send their utilities and probabilities to the arbitration module which, upon application of a multi-attribute utility estimation function, will ultimately determine the agent's optimal problem solving strategy, information action (if any), and base-level motor action.

2.5 Intelligent Agency — The Issues

An autonomous agent operating in a complex environment is constantly faced with the problem of deciding what action to take next. As Hanks and Firby [HF90] point out, formulating this problem precisely can be very difficult since it necessitates consideration of a number of informational categories which are often difficult to ascertain — for example, the benefits and costs to the agent of executing particular actions sequences; or which have been demonstrated from previous research to be problematic to represent — for example, models of agents' beliefs and desires about a world which is complex and unpredictable.

The *control problem* in an agent is the problem of deciding how to manage these various sources of information in such a way that the agent will act in a competent and effective manner. This problem, Hanks and Firby [HF90] suggest, amounts to balancing two "reasonable" approaches to acting in the world: the first, *deliberation*, involves making as many decisions as possible as far ahead of time as possible; the second approach, *reaction*, is to delay making decisions as long as possible, acting only at the last possible moment. At a glance, the first approach seems perfectly reasonable since, clearly, an agent which can think ahead will be able to consider more options and thus, with forethought, be more informed when deciding which action to take. On the other hand, since information about the future can be notoriously unreliable and, in many real-world situations, difficult or even impossible to obtain given the agents' changing time constraints, it would also seem reasonable that acting at the last moment should be preferred. In fact, except perhaps for a small number of special-case task domains, it would seem much more reasonable to assume that neither approach — deliberation or reaction — should be carried out to the full exclusion of the other.

As described in the previous chapter, this dissertation is concerned with the design and implementation of an integrated software control architecture — the TouringMachine architecture — suitable for controlling the actions of an autonomous agent operating in complex environments. Of particular interest in this dissertation is the integrated control of an autonomous, mobile agent capable of rationally carrying out a number of routine tasks in a complex multi-agent traffic domain — the *TouringWorld*.

The tasks or duties to be carried out by each agent in the TouringWorld are prioritised in advance by the agent's designer and include goals like avoiding collisions with other mobile agents and fixed obstacles, obeying a commonly accepted set of traffic regulations, and also relocating from some initial location to some target destination within certain time bounds and/or spatial constraints. Besides being limited in terms of its internal computational resources, each TouringMachine will start out with only limited knowledge of its world: in particular, although each TouringMachine possesses a topological map of the various paths and junctions defining all of the navigable routes in the world, it will have no prior knowledge regarding other agents' locations or goals or the obstacles it might encounter en route. In addition, each TouringMachine has limited means for monitoring and acquiring information from its surroundings and will be restricted in its capacity to communicate with other agents: intentions to turn or overtake are communicated via primitive signalling alone, much like a human driver does in a car.

The aim of this dissertation is to produce an integrated control architecture which will enable TouringMachines to carry out tasks and act on their environments autonomously and in accordance with a set of domain-specific evaluation criteria, namely, effectiveness, robustness, and flexibility. These criteria suggest a broad range of behavioural and functional capacities that each TouringMachine might need to possess:

- A TouringMachine should be capable of **autonomous** operation. Operational autonomy requires that the agent have its own *goals* and be able to select among these as and when required. In addition, as Covrigaru and Lindsay [CL91] argue, the agent should, among other things, be capable of interacting with its environment, be able to move (preferably fluidly) around its environment, have selective attention (this is also desirable since TouringMachines have limited computational resources), have a varied behavioural repertoire, and have differential responsiveness to a variety of environmental conditions.

- A TouringMachine should carry out its goals in an **effective** manner. Effective goal achievement requires that the agent be capable of carrying out its multiple tasks in an efficient and timely manner. Since among its various tasks, a TouringMachine must navigate along some route within a pre-specified time limit, the agent should be able to reason predictively about the temporal extent of its own actions. Also, because TouringMachines will operate in a partially-structured multi-agent world, they should, in order to complete their tasks, be able to *coordinate* their activities with other agents that they might encounter: that is, they should be capable of *cooperation*.[12]

- A TouringMachine should be **robust** to unexpected events. Successful operation in a real-time dynamic environment will require that TouringMachines be able to identify and handle — in a timely manner — a host of unexpected events at execution-time. For many events (such as the sudden appearance of a path-blocking obstacle) an agent will have little or no time to consider either what the full extent of its predicament might be or what benefits consideration of a number of different evasive manoeuvres might bring. In order to cope with such events, TouringMachines will need to operate with guaranteed responsiveness (for example, by using latency-bounded computational and execution techniques) as well as being fairly closely-coupled to their environments at all times. Since the time and location of such events will be unpredictable, TouringMachines will need to monitor their surroundings continually throughout the course of their goals.

- A TouringMachine should be **flexible** in the way it carries out its tasks. Due to the dynamic and unpredictable nature of the TouringWorld environment, and the fact that its multiple inhabitants must operate in real time with limited world knowledge, TouringMachines will inevitably be faced with various belief and/or goal conflict situations arising from unforeseen interactions with other agents.[13] Agents operating cooperatively in complex domains must have an understanding of the nature of cooperation. This, Galliers [Gal90] argues, involves understanding the nature and role of multi-agent conflict. To behave flexibly and to

[12]Following Bond and Gasser [BG88, page 19], cooperation in the TouringWorld is viewed simply as a special case of coordination among *non-antagonistic* agents. While TouringMachines are not actually benevolent (they are selfish with respect to their own goals and have the ability to drop or adopt different intentions according to their own preferences and situational needs) they are also not antagonistic since they do not intentionally try to deceive or thwart the efforts of other TouringMachines.

[13]Since TouringMachines operate in a multi-agent environment but perform actions purely for their own purposes (that is, they do not cooperate on a common plan), their interactions can be described as being merely *contingent* [Dav90, page 439].

> adjust appropriately to changing and unpredicted circumstance, TouringMachines should be designed to recognise and resolve unexpected conflicts rather than to avoid them. Also, for the purposes of control and coordination, TouringMachines must be able to reason about their own and other agents' activities. In this respect, each TouringMachine must have the capacity to *objectify* particular aspects of the world — that is, to construct and deploy internal models of itself and of other agents — to see where it fits in the coordinated process and what the outcomes of its own actions might be [BG88, page 25].

Although much of the above functionality could be described as deliberative (for example, reasoning about the temporal extent of actions, conflict resolution, reflexive modelling), it is unclear whether a strictly deliberative control approach based on traditional planning techniques would be adequate for successful operation in the TouringWorld domain. Most classical planners make a number of important simplifying assumptions about their domains which cannot be made about the TouringWorld : namely, that the environments remain static while their (often arbitrarily long) plans are generated and executed, that all changes in the world are caused by the planner's actions alone, and that their environments are such that they can be represented correctly and in complete detail. Given that the TouringWorld is dynamic and multi-agent and given that TouringMachines also have inherently limited physical and computational means for acquiring information about their surroundings, it seems clear that a strictly traditional planning approach to controlling TouringMachines would be unsuitable. Also, while it is true that planning systems capable of execution monitoring and interleaved planning and execution represent a significant advance on the earlier traditional planners, their usefulness in a highly dynamic and real-time domain like the TouringWorld is questionable, particularly given the reservations expressed in Section 2.2 above (and echoed by Georgeff [Geo90] and Bratman *et al.* [BIP88]) concerning their computational efficiency and inability to cope with situationally-varying time constraints.

Similarly, while the inclusion of at least some degree of non-deliberative control in TouringMachines would seem essential — particularly since the agents will need to be closely coupled to their environment, robust to unexpected events, and able to react quickly to unforeseen events and operate with guaranteed levels of responsiveness — it is questionable whether non-deliberative control techniques *alone* will be sufficient for providing TouringMachines with the complete behavioural repertoire necessary for successful operation in the TouringWorld environment. This argument deserves closer consideration.

The strength of purely non-deliberative architectures lies in their ability to identify and exploit *local* patterns of activity in their current surroundings in order to generate more or less hardwired action responses (using no memory, predictive reasoning, and only minimal state information) for a given set of environmental stimuli. Successful operation using this method of control presupposes: *(i)* that the complete set of environmental stimuli required for unambiguously determining subsequent action sequences is always present and readily identifiable — in other words, that the agent's activity can be strictly *situationally determined*; *(ii)* that the agent has no *global* task constraints — for example, explicit temporal deadlines — which need to be reasoned about at run-time; and *(iii)* that the agent's goal or desire system is capable of being represented *implicitly* in the agent's structure according to a fixed, pre-compiled ranking scheme.

Situationally determined behaviour will succeed when there is sufficient local constraint in the agent's environment to determine actions that have no irreversibly detrimental long-term effects. Only then, as Kirsh [Kir91] argues, will the agent be able to avoid representing alternative courses of actions to determine which ones lead to dead ends, loops, local minima, or generally undesirable outcomes. It follows, then, that if the agent's task requires knowledge about the environment which is not immediately available through perception and which can, therefore, only be obtained through some form of inference or recall, then it cannot truly be considered situationally determined. Kirsh [Kir91] considers several such tasks, a number of which are pertinent to the TouringWorld domain: activities involving other agents (as these often require making *predictions* of their behaviour and reasoning about their plans and goals [Dav90, page 395]); activities which require responding to events and actions beyond the agent's current sensory limits (such as taking precautions now for the future or when tracking sequences of behaviours that take place over extended periods of time); as well as activities which require some amount of reasoning or problem solving (such as calculating a shortest route for navigation).[14] The common defining feature of these tasks is that, besides requiring reliable and robust *local* control to be carried out, they also possess a non-local or *global* structure which will need to be addressed by the agent. For instance, to carry out a navigation task successfully in the TouringWorld an agent will need to coordinate various locally constrained (re-)actions such as slowing down to avoid an obstacle or slower moving agent with other more globally constrained actions such as arriving at a target destination within some pre-specified deadline.

[14] Kirsh also considers a set of non-situationally determined tasks which, while not carried out in the TouringWorld domain, nevertheless reflect key aspects of intelligent behaviour. These include activities which are creative and therefore stimulus free, such as much of language use, musical performance, mime, and self-amusement [Kir91, page 173].

While non-deliberative control techniques ensure fast responses to changing events in the environment, they do not enable the agent's action choices to be influenced by deliberative reasoning. In most non-deliberative architectures, the agent's goals are represented implicitly — in effect, embedded in the agent's own structure or behavioural rule set. When goals are not represented explicitly, Hanks and Firby [HF90] argue, they will not be able to be changed dynamically and there will be no way to reason about alternative plans for carrying them out.[15] Maes [Mae90] also argues that without explicit goals, it is not clear how agents will be able to learn or improve their performance.

Complex agents will need complex goal or desire systems — in particular, they will need to handle a number of goals, some of which will vary in time, and many of which will have different priorities that will vary according to the agent's situational needs. The implications of this, Kirsh [Kir91] argues, is that as agents' desire systems increase in size, there will be a need for some form of desire management, such as deliberation, weighing competing benefits and costs, and so on.

There are undoubtedly a number of real-world domains which will be suitable for strictly non-deliberative agent control architectures. It is less likely whether there exist any realistic or non-trivial domains which are equally suited to purely deliberative agents. What is most likely, however, is that the majority of real-world domains will require that intelligent autonomous agents be capable of a wide *range* of behaviours, including some basic non-deliberative ones such as perception-driven reaction, but also including more complex deliberative ones such as flexible task planning, strategic decision-making, complex goal handling, or predictive reasoning about the beliefs and intentions of other agents.

The thesis of this dissertation is that it is both *desirable* and *feasible* to combine suitably designed deliberative and non-deliberative control functions to obtain effective, robust, and flexible behaviour from autonomous, task-achieving agents operating in complex environments. The arguments put forward so far have attempted both to outline some of the broader functional and behavioural requirements for intelligent agency in complex task domains like the TouringWorld , and also to justify a hybrid control approach that integrates a number of deliberative and non-deliberative action control mechanisms. Remaining issues, such as how these particular requirements should be approached and designed for, and also how such design solutions are to be realised within the framework of a practical integrated architecture will be

[15] Note that these criticisms apply equally to systems whose goals are hardwired directly by the designer (for example, Pengi [AC87] or Creatures [Bro91b]) and to those whose goals are obtained through automatic compilation (for example, Kaelbling's situated automata [Kae91]).

the main focus of Chapters 3 through 6.

In addition to addressing the conjecture concerning the suitability of a hybrid control approach, the proposed control architecture is intended to address the importance of and need for extensive empirical evaluation of integrated agent architectures, not merely in terms of the per-agent, task-oriented criteria identified earlier in this section (effectiveness, robustness, and flexibility), but also in terms of the controlled agents' behavioural ecology [CGHH89] — that is, in terms of the functional relationships between agent design (agents' internal structures and processes), agent behaviour (the choice of tasks to be solved and the manner in which they are solved), and environmental characteristics. To address these issues, a highly parametrized and instrumented multi-agent simulation testbed has been implemented in conjunction with the TouringMachine control architecture. Enabling controlled, repeatable experimentation and facilitating the creation of diverse single- and multi-agent task scenarios, the TouringWorld Testbed is described in more detail in Chapter 7. Some preliminary experimentation with and analysis of the TouringMachine architecture is then presented in Chapter 8.

3

TouringMachines

'You see how it works, don't you? There's some sort of danger centering at the selenium pool. It increases as he approaches, and a certain distance from it Rule Three potential, unusually high to start with, exactly balances the Rule Two potential, unusually low to start with.'

Donovan rose to his feet in excitement. 'And it strikes an equilibrium. I see. Rule Three drives him back and Rule Two drives him forward—'

'So he follows a circle around the selenium pool, staying on the locus of all points of potential equilibrium. And unless we do something about it, he'll stay on the circle forever, giving us the good old runaround.'

Isaac Asimov, *Runaround*

3.1 Introduction

Operating in real-world domains means having to deal with multiple events at several levels of granularity — both in time and space. In these domains an agent will often need to perform complex tasks requiring it to pay some degree of attention to environmental change, temporal deadlines, computational resource bounds, and the impact the agent's shorter term actions might be having on its longer term goals. An agent, however, cannot be expected to model its surroundings in every detail as there will simply be too many events to consider, a large number of which will be of little or no relevance anyway.

To operate successfully in the chosen TouringWorld domain, an autonomous resource-bounded mobile agent must be both robust and flexible — it must be capable of carrying out its intended goals in the presence of dynamic, unpredictable events. To do this, the agent must be capable of exhibiting a range of different skills and behaviours. First, it will need to be reactive to deal with events which it may not have had sufficient time or resources to consider: since time will not stop or slow down for the agent to deliberate upon all possible courses of action for every world state, it must remain reactive if it is to survive long enough to complete its various tasks. Secondly, since one of the agent's main tasks in the TouringWorld will be to get from some starting location to some target destination within some specified time, it should be capable of rational goal-directed behaviour. And thirdly, since it will inhabit a world populated by other complex intentional entities — about which very little will be known in advance — it must be able to reason about what events are taking place around it, determine what effect these events could have on its own goals, and, within the real-time constraints imposed upon it by its task environment, select timely and appropriate action sequences which enable it to coordinate with any other agents that may be present.

What is required, in effect, is an agent architecture that can cope with uncertainty, react to unforeseen events, and recover dynamically from poor decisions. All of this, of course, on top of accomplishing whatever tasks the agent was originally assigned to do. Because the skills needed to operate in the TouringWorld have such disparate characteristics and requirements, the most sensible way of realising them, it will be argued below, is as separate activity-producing behaviours in a layered control framework. This has been the approach adopted in designing and implementing TouringMachines.

3.2 Overview

The TouringMachine agent architecture comprises three separate control layers: a *reactive* layer $\mathcal{R}$, a *planning* layer $\mathcal{P}$, and a *modelling* layer $\mathcal{M}$. The three layers (see Figure 3.1) are concurrently-operating, independently-motivated, and activity-producing: not only is each one independently connected to the agent's sensory apparatus (via the agent's *Perception Subsystem* — see Section 3.3.1 below) and has its own internal computational mechanisms for processing appropriate aspects of the received perceptual information, but they are also individually connected to the agent's effectory apparatus (via the agent's *Action Subsystem*) to which they send, when required, appropriate motor-control and communicative *action commands*.

Mediated by an enveloping *control framework* (see Figure 3.1 and Section 3.3 below), each of the TouringMachine's activity-producing layers is designed to model the agent's world at a different level of spatio-temporal abstraction and so is endowed with different task-oriented capabilities. The reactive layer $\mathcal{R}$, for instance, is intended to provide the agent with fast reactive capabilities for coping with immediate or short-term events which the agent's higher layers ($\mathcal{P}$ and $\mathcal{M}$) are either unaware of or have not had suf-

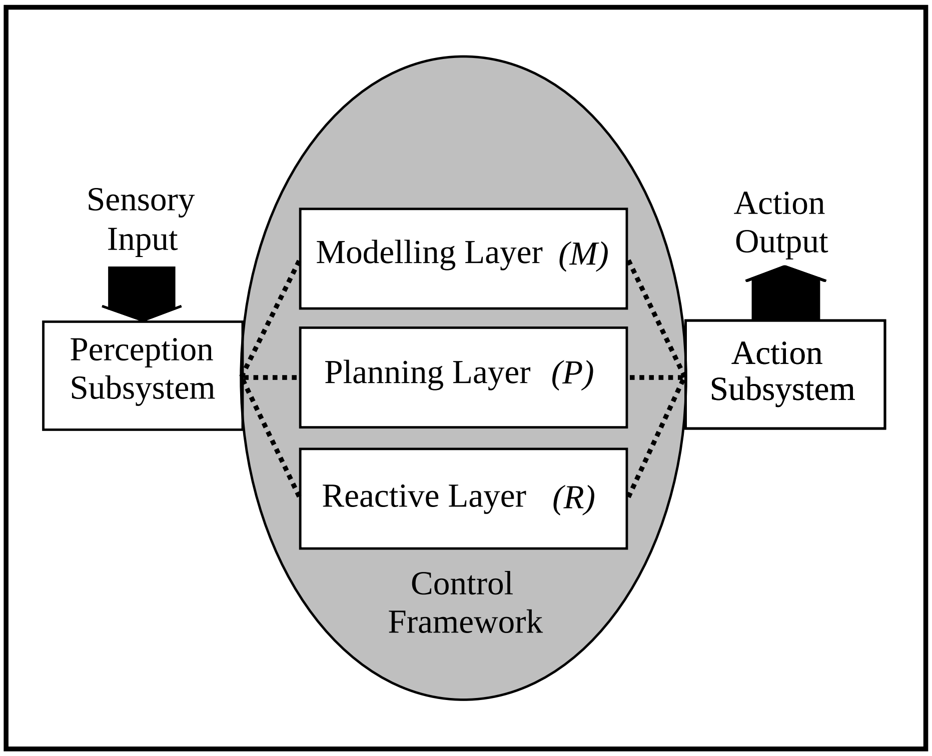

Figure 3.1: The TouringMachine agent control architecture.

ficient time or computational resources to compute suitable responses to. A typical event handled by layer $\mathcal{R}$, for example, would be the sudden appearance of a nearby and hitherto unseen agent or obstacle. Described in more detail in Chapter 4, layer $\mathcal{R}$ is designed to compute hardwired, domain-specific action responses to particular sets of environmental stimuli and thereby provide the agent with immediate feedback about various unexpected and potentially life-threatening events that can take place in the world.

The main purpose of layer $\mathcal{P}$, on the other hand, is to generate and execute plans of action which can help to achieve the agent's primary long-term domain task: that of relocating to a given target location within certain prespecified spatio-temporal constraints. Starting with a topological map of the various paths and path junctions in the world (but with no initial knowledge regarding any obstacles or other agents' whereabouts or intended actions), layer $\mathcal{P}$'s planner — the main functional component in this layer — makes use of a library of partially-elaborated, procedural plan structures to construct single-agent, linear route plans to the agent's desired destination. To cope with the fact that these "ideal" plans are likely to fail if the agent subsequently encounters any obstructing entities en route, layer $\mathcal{P}$'s planner, as described more fully in Chapter 5, has been realised as a hierarchical, partial planning system which can interleave plan formation and execution and defer committing to specific subplan execution methods until absolutely necessary.

While adopting such measures in layer $\mathcal{P}$ can help to minimise the need for corrective run-time *replanning* (this is also called *transformational planning*), situations will nevertheless arise in which the agent's initial plans fail to achieve their intended effect and where the only viable course of action will be to alter or dispose of some of its existing plans and then generate new ones which address the agent's newly acquired tasks and/or identified opportunities. Such situations are called *goal conflict situations* and they arise, in the TouringWorld domain, as a result of unexpected spatio-temporal interactions between two or more entities — for example, when the space-time trajectories of two agents intersect at a common point. Targeted at the levels of intra- and inter-agent goal coordination, the main purpose of a TouringMachine's layer $\mathcal{M}$ is to detect and foresee potential goal conflict situations — via execution monitoring (*observation*), abductive inference (*explanation*), and temporal and counterfactual reasoning (*prediction*) — and then to propose suitable courses of action which will hopefully preclude such conflicts. As described in Chapter 6, layer $\mathcal{M}$ provides an agent with facilities for incrementally building and saving *mental* or *causal models* of other world entities (and also of itself) which can enable it to "answer questions" about any entity's actions, dispositions, or motives [Min86, page 303]; for example, *"What are the entity's current intentions?"*, *"Which of the entity's goals is most threatened in the current conflict situation?"*, *"How will the entity resolve its most pressing goal conflict?"*, or *"What will the entity be doing T units of time from now?"*

Collectively, the three control layers $\mathcal{R}$, $\mathcal{P}$, and $\mathcal{M}$ are aimed at providing a TouringMachine agent with a variety of deliberative and non-deliberative task-achieving behaviours; these include behaviours that are situationally determined or reactive, goal-directed, reflective, and also predictive. However, because layers operate concurrently, are activity-producing (that is, each layer can independently send action commands to the agent's Action Subsystem), and are designed so that each addresses a different (and therefore limited) aspect of the agent's necessary behavioural repertoire, it is inevitable that, from time to time, one layer's proposed actions will conflict with those of another. Layers, in effect, are *approximate machines*, and as a result, need to be mediated by an enveloping control framework if the agent, as a single whole, is to behave appropriately in each different world situation.

3.3 The Control Framework

Like the behaviours employed in the agent architectures of Kaelbling [Kae87] and Brooks [Bro86] or like the three control components used in the Entropy

Reduction Engine (ERE) architecture [BD90], each of the three TouringMachine control layers exhibits a more or less independent ability to carry out its own assigned tasks. In the TouringWorld domain, in particular, this includes reacting to immediate threats and short-term events (layer $\mathcal{R}$), planning time-constrained routes to specified locations (layer $\mathcal{P}$), and detecting and resolving ongoing conflicts within the agent's current goal set (layer $\mathcal{M}$). As Bresina and Drummond [BD90] point out, however, independent ability alone cannot guarantee appropriate performance: a layer operating in isolation will typically exhibit fairly poor performance (each TouringMachine layer, after all, is an approximate machine dealing only with a subset of the agent's complete goal set) and thus its performance will only improve through suitably designed interactions with the agent's other layers. Such interactions form the basis of the TouringMachine control framework.

The primary purpose of the TouringMachine control framework, then, is to ensure the agent behaves appropriately in each different world situation. It attempts to do this by taking into account the agent's changing situational and task-related needs. As shown in Figure 3.2, a TouringMachine's internal control framework consists of three major functional components: the *Perception and Action Subsystems* through which each layer interfaces to the external environment, a facility for *inter-layer message passing* enabling layers at run-time to exchange certain types of control information, and a set of context-activated mediatory *control rules* which, by effecting modifications to the inputs to and outputs from any of the agent's layers, can be used to resolve any action conflicts which might arise as a result of the limited functional and behavioural scope of each of the agent's three control layers. Each of these aspects of the control framework will now be described in more detail.

3.3.1 Perception and Action Subsystems

Being planning agents, TouringMachines are able to construct predictive models about a number of different future world states and about the potential effects that its planned actions, if carried out in full, should have in these states. However, while the ability to predict can often prove a very useful asset, at other times, such as when the agent is operating in a rapidly changing world or when the agent has only limited initial knowledge about its task environment, any predictions that are made could ultimately prove very unreliable. As Dean and Wellman [DW91, pages 9–10] suggest, one way an agent can overcome — or at least partially offset — the effects of imprecise predictive models is to rely on such models for short-term predictions only, and to supplement the agent's control function with feedback obtained through frequent sensing of the environment which can then be used to correct any potential

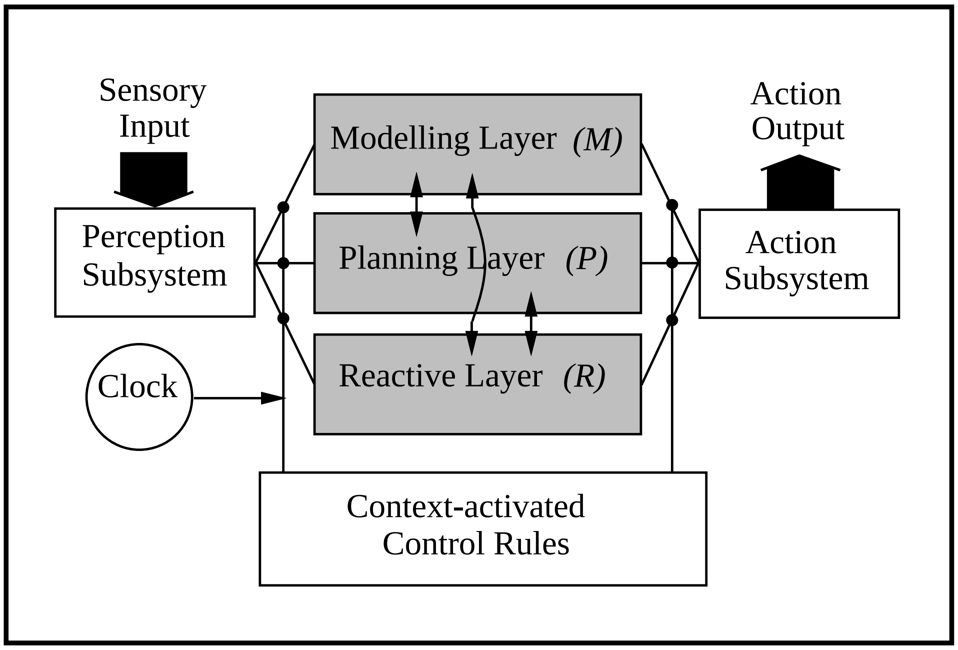

Figure 3.2: A TouringMachine's mediating control framework incorporating Perception and Action Subsystems, inter-layer message passing, and context-activated mediatory control rules.

errors in its longer term predictions.

The Perception and Action Subsystems are aimed at providing TouringMachines with the necessary input-output or *sensorimotor control* capabilities for sensing and acting in a dynamically changing world. In the TouringWorld domain such control capabilities must include the perception of any entities which are initially unknown in the world (for example, other TouringMachines , traffic lights, or obstacles), as well as the effecting of various throttle (`change-speed`), steering (`change-orientation`), and communicative action commands (for example, `signal-left`, `honk-horn`, or `flash-headlights`).

Similar to the approach used in Kaelbling's agent design [Kae87], inputs to and outputs from a TouringMachine are generated cyclically in a *synchronous* fashion, the start and finish times of each synchronous input-output processing cycle — or *timeslice* — being established with the use of an internal agent *clock* (see Figure 3.2).[1] A timeslice can be regarded as the basic unit of agent processing activity: at the start of each timeslice the agent receives sensory input via its Perception Subsystem; at the end of each timeslice the agent

[1]Although the clocks used for synchronising input-output processing are internal to each TouringMachine agent, they are in fact initialised by another clock which is global to the entire TouringWorld environment. As described in Section 7.4.2, this common world clock is operated and maintained by a TouringWorld simulator process called WorldUpdater.

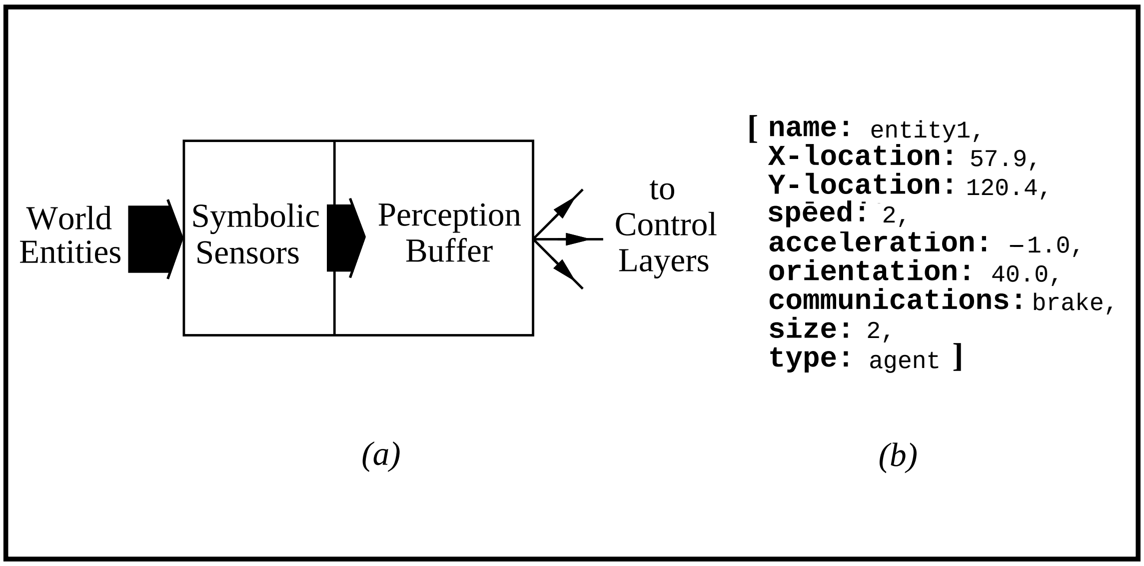

Figure 3.3: A TouringMachine's Perception Subsystem *(a)* and an example sensory item that would be stored in the subsystem's Perception Buffer *(b)*.

effects any required actions on the world via its Action Subsystem; in between these two stages, the agent — or more precisely, each one of the agent's three independent and concurrently-operating control layers — is charged with the task of processing the information produced by the Perception Subsystem and, if required, submitting suitable action commands to the agent's Action Subsystem.[2] The specific computational mechanisms each layer uses in processing its sensory input and generating action commands are not important at this point and will instead be detailed in the next three chapters. The operations of the Perception and Action Subsystems do, however, deserve further description.

As shown in Figure 3.3 *(a)*, a TouringMachine's Perception Subsystem comprises two main components: a set of symbolic sensors with which to acquire items of information describing the current environmental layout, and a *Perception Buffer* which is used for storing the perceptual information items recently received from these sensors.

TouringMachines are intended for use in a simulated environment, the TouringWorld , in which everything can be regarded as a discrete symbolic entity. Via its sensors, then, a TouringMachine is considered capable of uniquely identifying each different world entity (that is, it is considered ca-

[2]The length of a timeslice is expressed as a positive real number of (simulated) world seconds and is defined via a TouringWorld testbed *parameter* called **WorldTimeIncrement**. Along with with the concept of the simulated world clock, the notions of timeslice and timeslice length are described more fully in Section 7.4.2.

pable of assigning a unique symbolic name to each separately perceived entity), as well as being able to recognise certain physical properties of each entity: namely their current (x, y) location[3], speed, acceleration, orientation, observable communicated signals, dimensions, and type (the latter enabling it to distinguish, for instance, whether the entity is a TouringMachine , obstacle, traffic light, path junction, wall, kerb, lane marking, or path-side information sign). In this respect, TouringMachines ' sensory capabilities resemble those of most other agents deployed in simulated environments [CGHH89, DM90, PR90, SH88, VB90, Woo90].

Each time a given world entity has been fully identified by a TouringMachine's sensors, a *multi-attribute information record* is created for it which is then submitted for storage to the agent's Perception Buffer. Each stored information record (see Figure 3.3 *(b)*, for example) can be regarded as a "snapshot" of the current *physical extent* of a particular world entity, and the Perception Buffer's entire collection of records can consequently be regarded as a momentary snapshot of the TouringMachine's external observable environment.

Of course, being resource-bounded agents, TouringMachines cannot be expected to perceive every entity that might be present in the world. In fact, since TouringMachines are intended for use in real-time environments, it is important to ensure that the combined latencies of each of their internal functional operations be guaranteed *constant-bounded* (see Section 3.6 below). As far as a TouringMachine's sensing operations are concerned, this effectively means placing a limit on the total number of entities that may be sensed during any given timeslice. In TouringMachines this can be approximated by placing an upper bound on the agent's spatial sensing range.[4]

A TouringMachine's Action Subsystem is shown in Figure 3.4. This consists of a limited-capacity *Action Buffer* for receiving the motor-control and communicative action commands sent by the agent's three control layers ($\mathcal{R}$, $\mathcal{P}$, and $\mathcal{M}$), and a set of effectors which are capable of translating such commands into corresponding physical actions on the agent's world.[5] The

[3]Entity locations in the TouringWorld are treated as real-valued two-element points on a global Cartesian coordinate grid (see Section 7.3).

[4]In fact, as described more fully in Section 7.4.3 (and also illustrated on page 146), range is merely one of several user-level properties or parameters that can be used to "customise" TouringMachines' sensors. Other parameters include the frequency with which the agent senses the world (**SensingRate**) and the specific sensing algorithm that it uses (**SensingAlgorithm**) — the latter dictating, for example, whether the agent can sense occluded objects and/or whether it can sense objects that are behind it as well as those that are in front of it. For present purposes, these parameters can be ignored; they will, however, be returned to when describing the TouringWorld simulator in Chapter 7.

[5]Of course, since TouringMachines operate in a simulated environment, their physical

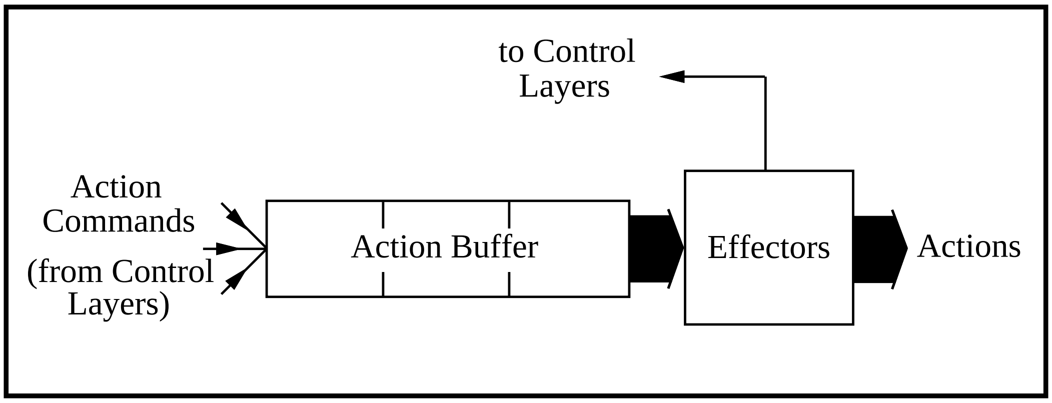

Figure 3.4: A TouringMachine's Action Subsystem.

complete set of TouringMachine action commands is given in Figure 3.5.

A TouringMachine's control layers are designed to submit action commands in response to the agent's changing situational and task-related needs. Which of these needs any given layer will decide to respond to will ultimately depend on each layer's internal programming. For instance, a TouringMachine's reactive layer $\mathcal{R}$ might submit an orientation-changing action command if the agent's sensors have detected the presence of an obstacle which is considered (by layer $\mathcal{R}$) to be too close to ignore; at the same time, a TouringMachine's planning layer might submit a different action command if, for example, the route plan that layer $\mathcal{P}$ is currently executing stipulates that the agent change direction at an upcoming path junction. Regardless of why a particular layer might choose to submit an action command, though, it should be understood that, during any given timeslice, any layer — $\mathcal{R}$, $\mathcal{P}$, or $\mathcal{M}$ — is capable of doing so.

To simplify the programming of a TouringMachine's Action Subsystem and, perhaps more importantly, to ensure the agent remains responsive to any important events that might take place in the environment while the agent is processing its sensory input, the TouringMachine architecture has been constrained in two ways. Firstly, each activity-producing control layer has been designed to submit *at most* one action command per input-output processing cycle; in other words, if during a given timeslice a layer decides to submit an action command to the Action Buffer, it will do so and then immediately suspend all of its operations until the start of the next timeslice. Secondly, the TouringMachine's effectors have themselves been constrained to effect one action (or zero if no action command happens to reside in the Action Buffer)

actions are in fact mimicked with the use of one of the TouringWorld simulator processes — WorldUpdater — which is designed to maintain an up-to-date and physically plausible model of the agents' world (see Section 7.4.2).

```
    change-speed(S)           S: FLOAT
    change-orientation(O)     O: ORIENTATION
    signal-left(SL)           SL: STATUS
    signal-right(SR)          SR: STATUS
    flash-headlights(FH)      FH: STATUS
    honk-horn(HH)             HH: STATUS
    hazard-lights(HL)         HL: STATUS
    fog-lights(FL)            FL: STATUS
```

where STATUS $\in$ {on, off} and ORIENTATION $\in$ [0.0, 360.0).

Figure 3.5: Motor-control and communicative action commands available to TouringMachines operating in the TouringWorld.

during any given timeslice. Of course, given that each of the *three* layers can independently submit an action command and that only *one* of these may be effected during any given timeslice, this obviously raises the question of which action command should be selected. This, in fact, is one area where the TouringMachine's mediatory *control rules* come into play. Before discussing these, however, some final comments regarding TouringMachine actions and effectors should be made.

For the sake of creating as realistic a simulation of mobile agent control as possible, a number of physical and kinematic control constraints have been imposed on TouringMachine agents: these include limits on their maximum speed, rates of acceleration and deceleration, as well as the rate at which they are able to change orientation (in other words, the maximum angle through which they are able to turn during a single timeslice).[6] Whenever a TouringMachine's effectors are faced with executing a new action command, these control constraints are used to determine whether the corresponding action can be carried out in full as intended and, when not, to prompt the TouringMachine's effectors to send a message to the appropriate control layer — in other words, the layer that originally submitted the "offending" action command — explaining what "problem" has just occurred.[7] In fact, regardless of whether

[6] In fact, these constraints have been implemented as TouringWorld Testbed parameters (**MaxSpeed**, **MaxAcceleration**, **MinAcceleration**, and **MaxTurningRate**, respectively) and so will be re-visited in Chapter 7.

[7] Two such "problems" can be identified with TouringMachines at execution time: when attempting to effect a `change-speed` action that causes the agent to exceed its maximum speed, and when effecting any action that cannot be completed within a single processing cycle and therefore requires iteration over two or more processing timeslices. Both of these

an action is deemed problematic or not, each time any action is executed, the TouringMachine's effectors will automatically send a message to whichever layer submitted the action command that was selected, essentially to confirm the fact that its command was the one accepted for execution. The purpose of such feedback is primarily to enable the agent's control layers to distinguish between *planned* actions (those submitted by layers $\mathcal{P}$ or $\mathcal{M}$) and *unplanned* actions (those submitted by layer $\mathcal{R}$). As discussed in Chapter 6, this is useful when a layer has to decide whether to re-try its failed action attempts.

3.3.2 Control Rules: Suppressors and Censors

A consequence of TouringMachines employing three concurrently-operating activity-producing control layers that have independent access to the agent's sensors and effectors and that function with only partial views of the agent's external surroundings, is that, from time to time, the actions taken by these layers will conflict.

In the TouringMachine architecture two types of inter-layer conflicts can be identified: *(i)* those which result from two or more layers perceiving and addressing a common event in the environment (for example, when layer $\mathcal{R}$ and layer $\mathcal{M}$ each propose separate actions to avoid the same obstacle); and *(ii)* those which result from two or more layers addressing different events in the environment (for example, when layer $\mathcal{P}$ suggests an action to move in the direction of the agent's target destination and layer $\mathcal{R}$ suggests an evasive maneuver to avoid colliding with a nearby obstacle). Resolution of such conflicts requires some form of mediation. In the TouringMachine architecture this is achieved with the use of context-activated mediatory *control rules*.

Acting as filters both between the agent's sensors and its control layers, and between the agent's control layers and its effectors, control rules are applied, in parallel, once at the beginning and once at the end of each input-output processing timeslice (see Figure 3.2). In fact, not all rules are applied at both synchronisation points. There are two types of control rules: those which are applied exclusively at the beginning of each timeslice and those which are applied exclusively at the end of each timeslice. The rules applied at the beginning of a timeslice are called *censor rules* and these have the effect of filtering (censoring) selected information (sensory items) from the inputs to the three control layers. The rules applied at the end of a timeslice are called *suppressor rules* and these have the effect of filtering (suppressing) selected information (action commands) from the outputs of the three control layers.

Both types of control rules are of the *if-then* condition-action type. In

will be discussed in more detail in Chapters 5 and 6.

```
censor-rule-1:
    if entity(obstacle-6) ∈ Perception-Buffer
    then
       remove-sensory-record(layer-R , entity(obstacle-6))

suppressor-rule-3:
    if action-command(layer-R -rule-6*,
                      change-orientation(_)†) ∈ Action-Buffer
       and
       current-intention(start-overtake)
    then
       remove-action-command(layer-R , change-orientation(_)) and
       remove-action-command(layer-M , _)
```

* layer-$\mathcal{R}$ -rule-6 is the reactive (layer $\mathcal{R}$) rule which is invoked in order to avoid crossing a path lane marking (see Section 4.3).

† "_" denotes a don't-care or *anonymous* variable [CM81] (see also the comment about Prolog unification in Section 5.4.2).

Figure 3.6: Two example control rules: censor-rule-1 and suppressor-rule-3.

the case of censor rules, the conditional parts are conjunctions of statements that test for the presence of particular sensory records in the agent's Perception Buffer. Censor rules' action parts consist of operations to prevent particular sensory records from being fed as input to *selected* control layers. In Figure 3.6, for example, the censor rule censor-rule-1 is used to prevent layer $\mathcal{R}$ from perceiving (and therefore, from reacting to) a particular obstacle which, for instance, layer $\mathcal{M}$ might have been programmed to deal with. In the case of suppressor control rules, conditional parts are conjunctions of statements which, besides testing for the presence of particular action commands in the agent's Action Buffer, can also test the truth values of various items of the agent's *current internal state*.[8] Suppressor rules' action parts consist of operations to prevent particular action commands from being fed through to the agent's effectors. In Figure 3.6, for example, the suppressor rule suppressor-rule-3 is used to prevent layer $\mathcal{R}$ from reacting to (steering away from) a lane marking object whenever the agent's current intention is to overtake some other agent that is in front of it.

[8]Internal TouringMachine state, as discussed later in Chapter 6, comprises statements about the agent's *current* (present timeslice only) beliefs, desires, and intentions.

Any number of censor control rules can fire (and remove selected control layer input) when these are applied at the beginning of a timeslice. Suppressor control rules, on the other hand, are assumed to have been crafted by the agent's designer in such a way that: *(i)* at most one will fire in any given situational context (an agent's situational context is taken to be the combination of its perceptual input set and its current internal state), and *(ii)* at most one action command will remain in the Action Buffer after the appropriate suppressor rule's action part has been executed. In Figure 3.6, for example, the two `remove-action-command` operations in the action part of suppressor control rule `suppressor-rule-3` will ensure that at most action command — in this case, that sent by layer $\mathcal{P}$ — will remain in the agent's Action Buffer. By crafting suppressor control rules in this way, a TouringMachine's effectors can be guaranteed to receive no more than one action command during any given timeslice.

In principle, a TouringMachine's suppressor and censor control rules can be seen to operate in a manner similar to Minsky's *suppressor-agents* and *censor-agents*, respectively. In Minsky's Society of Mind framework, suppressor-agents are those which wait until you "get a bad idea" — which in the case of TouringMachines, is equivalent to submitting a "bad" action command — and then prevent you from taking the corresponding action. Censor-agents, on the other hand, do not wait until a certain bad idea — or "bad" action command — occurs; instead, they intercept the "states of mind" — or in the case of TouringMachines, perceptual inputs — that usually precede the idea [Min86, page 275]. Application of a TouringMachine's suppressor and censor control rules also approximates the use of output signal *inhibition* and input signal *suppression*, respectively, in Brooks' subsumption architecture [Bro86] (see also Section 2.3.1).

3.3.3 Inter-layer Message Passing

The main attraction of the TouringMachine mediatory control rule framework just described is that its operation is more or less "transparent" to the agent's three control layers: each layer can essentially act as if *it alone* were controlling the agent's activities, remaining largely unaware of any "interference" — either by other control layers or by the censor and suppressor rules of the control framework — with its own inputs and outputs. The potential advantages of such a control scheme include increased operational concurrency, robustness, as well as enhanced programmability and testability gained through the use of highly modular design components (more on this topic below).

TouringMachine control layers, nevertheless, are approximate machines addressing isolated aspects of the agent's full behavioural repertoire; as a result, layers will invariably be ill-equipped to deal with each and every one of the agent's evolving task demands. One way to overcome this "isolation" problem is by enabling layers to assist with each others' operations through the exchange of control information concerning relevant aspects of the agent's changing task-related requirements or situational context. For this purpose, an inter-layer messaging scheme is included in the TouringMachine control framework.

The messages which are exchanged between control layers all agree with the simple format:

$$\langle Receiver, Sender, Time, Body \rangle$$

where $Receiver$ and $Sender \in \{\texttt{layer-}\mathcal{R},\ \texttt{layer-}\mathcal{P},\ \texttt{layer-}\mathcal{M}\}$, $Time$ is the time (according to the agent's internal clock) when the message was originally sent, and $Body$, the content of the message, is some application- or layer-specific pattern which can be interpreted — and subsequently acted upon — by the receiving control layer.

Messages can be classified into two types: those which are used in a *passive* way, simply to convey potentially relevant information to some other layer (for example, when layer $\mathcal{R}$ offers suggestions to layer $\mathcal{M}$ about which world entities the latter might want to focus its attention on — see Section 4.2); and those which are used in an *active* way, to alter another layer's control decisions (for example, when layer $\mathcal{M}$ sends instructions to layer $\mathcal{P}$ outlining a new task it must generate a plan for — see sections 5.4.2 and 6.5.4). Precisely when and for what purpose particular messages are exchanged between layers is very much tied in with the lower level operations of the individual agent control layers. As such, further discussion of the TouringMachine inter-layer message passing mechanism will be deferred to the next three chapters.

3.4 TouringMachines — A Layered Approach

The principal feature of the TouringMachine agent architecture is that it integrates a variety of deliberative and non-deliberative control functions. The main reason for advocating such a *hybrid* control approach is to attempt to combine into a single framework some of the advantages most often associated with purely deliberative architectures — namely, reasoned action choice and flexible long-term goal handling — with some of those which are more typical of purely non-deliberative ones — in particular, operational robustness, distribution of control, real-time responsiveness, and closer coupling of perception to action.

Two broad categories of hybrid agent control architectures can be distinguished: *uniform* architectures, which employ a single representation and control structure for both action and deliberation, and *layered* architectures, which use different representations and algorithms to perform these functions in a number of different layers. As Hanks and Firby [HF90, page 67–68] argue, the division between uniform and layered approaches reflects "a bias" as to which problems the architectures' implementors wish to address. Uniform architectures such as PRS [GLS87, GI89] or GUARDIAN [HR88, HR90], for example, make the assumption that action and deliberation are so closely related that these cannot usefully be handled separately.[9] By design, uniform architectures can easily bring all of their deliberation machinery to bear on every individual action decision that has to be made; however, because of the considerable computational cost of doing this, any agents that were controlled this way would soon fail to react in a guaranteed timely manner. To cope with this problem of responsiveness, uniform architectures need to provide means for making explicit control decisions about when the agent should act versus when it should deliberate further. While most uniform architectures have addressed this problem by allowing agents to include their control decisions explicitly within their internal plan representation and by allowing them to reason about their own reasoning methods (using meta-KAs in PRS [GLS87] and control plans in GUARDIAN [HR88], for example), care needs to be taken to ensure that such control frameworks avoid an infinite regress of metareasoning.

Layered architectures, on the other hand (for example, APE [SH90], ERE [BD90] and Phoenix [CGHH89]), all make the basic assumption that the agent's reaction time is so critical and that deliberation is so slow that the agent will often need to act without resorting to any deliberation at all. Because of such timescale mismatches, then, deliberation and action in layered architectures are separated into distinct modules using different control methods and possibly also different world models or plan representations [HF90, page 68].

To an agent operating in the TouringWorld domain, a number of events at several levels of spatio-temporal granularity can take place at any given point in time: for example, the sudden appearance of an obstacle within immediate collision range, the observed changing of colour of an upcoming traffic light, or the projected failure to arrive at some yet-to-be-observed target destination within the agent's pre-specified time bounds. Successful operation in the TouringWorld, it would therefore seem, will require that agents be capable of

[9]In fact, since neither of these particular systems make any specific commitments as to when and how deliberation and action should be interleaved, Hanks and Firby [HF90, page 67] prefer to refer to these systems simply as "frameworks" for encoding agent architectures rather than as actual agent architectures.

fulfilling a number of pre-specified goals *while simultaneously* responding to a number of dynamic events and making timely and appropriate short- and/or long-term control adjustments when any of their pre-specified goals appear threatened.

Following Hanks and Firby [HF90], among others, the design methodology adopted for the agent control architecture proposed in this dissertation reflects the belief that, since short-term acting is driven primarily by urgency but longer term deliberation requires (often substantial amounts of) time in order to consider alternative action choices and their possible consequences, different reasoning methods and representations will be appropriate for acting and deliberating. The TouringMachine architecture, therefore, adopts a layered approach to the agent control problem.[10]

As with the non-deliberative architectures of Brooks [Bro86] and Kaelbling [Kae87], overall control in the TouringMachine architecture is divided across a number of *vertical* layers — at present, three — each of which is designed to achieve a particular level of task-oriented competence.[11] Like Brooks' subsumption levels, a TouringMachine's control layers are designed so that if a higher layer (for example, $\mathcal{M}$) ceases to function, one or more of its lower layers (for example, $\mathcal{R}$) will still continue to provide the agent with some degree of competence, albeit considerably limited. This is made possible by ensuring that control layers have continuous and independent access to the agent's perceptual and effectory machinery.

Like Ogasawara's behaviour modules [Oga91] (and unlike Brooks' subsumption levels [Bro86]), higher level TouringMachine control layers need not necessarily subsume all of the operations of the agent's lower level layers. TouringMachine layers are designed to work independently — but also cooperatively — on complementary aspects of the agent's overall task load: layer $\mathcal{R}$ is designed to handle short-term events and immediate threats, layer $\mathcal{P}$ is designed to handle basic task planning duties, while layer $\mathcal{M}$ is intended to deal with plan conflicts arising from unforeseen multi-agent conflicts. In the TouringMachine architecture, no individual layer is capable of addressing each

[10]There are a number of other reasons for advocating a layered control approach, including increased behavioural robustness and operational concurrency, as well as improved program comprehensibility and system testability and analysability. These points have been argued elsewhere, most notably by Brooks [Bro86, Bro91b], Spector [SH90], and Davis [Dav90, page 312].

[11]While it is true that a certain amount of *horizontal* (functional) decompostion takes place within some of the TouringMachine layers (this, perhaps, being most evident when looking at the internal workings of layer $\mathcal{M}$ — see Chapter 6), the primary decomposition of the agent's control problem is *vertical* — in other words, divided among several activity-producing or task-achieving modules "on the basis of desired external manifestations of the [agent] control system." [Bro86, page 16].

and every one of the agent's situational and task-related needs; thus, if any task-level "intelligence" could ever be attributed to a particular TouringMachine, this would simply be the result of a carefully programmed combination of its three component control layers. In this respect, TouringMachine control layers are quite similar — in principle, at least — to the *proto-specialist* agencies in Minsky's Society of Mind framework: separate (sub-)agencies sharing a common set of sensory and effectory "organs" with each proto-specialist addressing a different basic need of the (super-)agent [Min86, page 165].

As much as possible, the division of labour among a TouringMachine's three activity-producing control layers is intended to reflect the different levels of spatio-temporal granularity which characterise the TouringWorld task environment. However, while in this sense, a TouringMachine's control layers could be described as constituting a hierarchy — the representations for world modelling used by its three layers appear increasingly abstract as one moves from layer $\mathcal{R}$ up to $\mathcal{M}$ — a similar characterisation cannot readily be made about the flow of control between the layers. Like Brooks' subsumption architecture [Bro86] (and unlike Spector and Hendler's APE design [SH90]), there is no strict hierarchical or prioritised flow of control between the different layers of the TouringMachine architecture. Rather than consisting of a set of high-level layers which try to account for the "big picture" either by issuing general advice to faster low-level layers [DW91, page 470] or by calling such lower level layers as mere subroutines, TouringMachine layers operate concurrently, proposing actions independently of each other's operational activities. It is not the case, therefore, that a TouringMachine's layer $\mathcal{M}$ will always be favoured over its layer $\mathcal{P}$ or that its layer $\mathcal{P}$ will always be favoured over its layer $\mathcal{R}$; instead, whichever action an agent ends up taking ultimately depends on the agent's situational context, which, defined in terms of its current observations, beliefs, desires, and intentions, is taken into consideration at execution time through application of the agent's suppressor control rules. Thus, although TouringMachines can essentially be described as *planning agents*, the layered control framework employed by TouringMachines acknowledges the potential unreliability of predictive plans by ensuring that their plans are treated simply as *resources* for action rather than as strict *recipes* for controlling every aspect of the agent's behaviour. This issue will be touched upon again in Chapter 8.

3.5 TouringMachines — Resource-boundedness and Rationality

A perfectly rational agent would always bring all of the information in its memory and in its environment to bear on its various perception, decision-making, and execution tasks. Ideally one would wish for an agent control architecture which, on the one hand, was capable of generating rational, minimal, and provably correct action sequences, and on the other, was capable of flexibly and robustly dealing with the real-time pressures that are characteristic of dynamic multi-agent domains.

In toy domains like the blocks worlds of classical AI planning fame, it may well be possible for a single agent to guarantee the generation and execution of optimal action sequences. In realistic domains, however, the requirements to behave at once correctly, flexibly, *and* robustly conflict with each other, agents often having to rely on heuristic or satisficing methods of decision-making to ensure the successful completion of their tasks. As Simon puts it: the agent will have a choice between 'optimal decisions for an imaginary simplified world or decisions that are "good enough," that satisfice, for a world approximating the complex real one more closely.' [Sim81, page 35].

The TouringMachine architecture is aimed at supporting the development of real-time embedded agents. In such agents, there is not always enough time — nor indeed computational power — to make use of all available knowledge for each and every task-related activity. To cope with such constraints, TouringMachines are designed to make use of latency-bounded heuristic functions to simplify some of their decision-making processes. For example, TouringMachines satisfice with respect to planning by employing a limited search that is directed by stored heuristic knowledge. Also, TouringMachines' layers $\mathcal{P}$ and $\mathcal{M}$ are designed so as not to process all of the sensory input collected by their Perception Subsystems; instead they make use of a selective attention mechanism which helps them to focus only on those environmental features which are deemed relevant to the agent's situational and task-related needs. A consequence of this, however, is that, at times (for instance, in sufficiently time-constrained situations), TouringMachines may well generate plans which are less than optimal, some even failing to achieve their initially intended tasks unless appropriately repaired.

Another complication which arises vis-à-vis guaranteeing rational behaviour is the fact that TouringMachines have multiple goals, some of which, on occasion, will conflict with one another. For example, the act of slowing down to avoid colliding with another agent will likely conflict with the agent's main time-constrained navigational task, especially if this task's deadline was reasonably tight to begin with. Similarly, if an agent waiting at a red light determines that it is about to be hit from behind by another entity, a TouringMachine will, by design, reactively move to avoid the collision — even if this results in driving through the red light.[12] So, because a TouringMachine will not be able to attend to all of its goals simultaneously, some of its actions may occasionally be inconsistent with one or more of these goals. Note, however, that such actions typically serve a protective purpose for the agent and so should not be regarded as undesirable. On the contrary, since in dynamic domains agent robustness and survival are usually considered more important than correctness, protective actions — perfectly rational or otherwise — must surely be considered indispensable.

According to Bratman *et al.* [BIP88], a rational agent is one which is *committed* to doing what it plans; and as such, only under exceptional circumstances — for example, when an explicit impediment is detected or when some task delay can be foreseen — should the agent undertake to alter its plans. Knowing when to reconsider committed plans is not simple: as Minsky [Min86, page 163] points out, "Too much commitment leads to doing only one single thing; too little concern produces aimless wandering." What this suggests, in fact, is the existence of a "tension" between the *stability* that an agent's plans must have in order to provide a focus for the agent's deliberative reasoning processes, and the *revocability* that the same plans must also exhibit, given that they will only ever have been conceived with partial information about the agent's past, present, and future states.

Long-term rational behaviour, then, would appear to result from continually balancing the tension between plan stability (goal-orientedness, future-directedness) on the one hand, and plan revocability (reactivity, flexibility) on the other. As Maes [Mae90] suggests, obtaining the right balance of such behavioural characteristics will depend on particular aspects of the agent's task and environment: for example, the precision with which the task must be carried out, the time available for making control decisions, or the degree of predictability in the agent's surrounding environment. In fact, as shown in Chapter 8 — and confirmed by several other investigations into resource-bounded agency [BIP88, SH88, CGHH89, HHC90, PR90] — the production of rational behaviour can also be seen to depend on characteristics of the agent's own internal design or configuration — for example, the agent's degree of sensitivity to unanticipated environmental change.

[12]As will be explained in Chapter 6, a TouringMachine builds and executes run-time plans to stop at red lights in order to satisfy `obey-regulations` — one of several layer $\mathcal{M}$ goals. In this example, then, the action of going through a red light will conflict with the agent's goal `obey-regulations`.

3.6 TouringMachines — Real-time Activity

Successful operation in a dynamic environment like the TouringWorld will require real-time responses to a range of unanticipated events and planning exceptions. Real-time response might be required, for example, if an agent is to avoid missing an important task deadline or, more importantly, if it is to prevent the catastrophic consequences of colliding with another agent or obstacle. In fact, as is most often the case when operating in complex environments, agents will typically carry out a mixture of different tasks, some of which will be characterised in terms of *hard deadlines* — those which are time-critical and have to be met at all costs — and others which are characterised by *soft deadlines* — those which, if not met, will not result in a compromise to the agent's integrity.

Conventional real-time systems are designed to meet the individual timing requirements of a set of system tasks in such a way that they are not only logically predictable, but also temporally predictable — particularly when operating in peak-time or worst-case conditions. For such real-time systems to be successful — that is, for these systems to be both fast *and* predictable — they must be capable of scheduling their tasks in such a way that all highly critical tasks can always be guaranteed to meet their own pre-specified timing constraints. For this to occur, such a system must ensure, in advance, that sufficient resources can be pre-allocated for achieving each and every one of its time-critical tasks [SR88].

For a resource-bounded agent operating in a dynamic, multi-agent (and thus inherently unpredictable) world like the TouringWorld, the conception of real time needs to be revised somewhat. Whereas conventional real-time systems are usually defined in terms of a statically determined control sequence (often programmed as a fixed decision tree), TouringWorld agents, in order to interact robustly with other agents which are present in their world, have to be capable of dealing with both hard and soft *aperiodic* real-time events which might occur at unexpected times throughout the period of their operation.[13]

To cope with such aperiodic events (for example, the sudden appearance of another agent from a nearby side path), TouringMachines make use of an opportunistic control strategy — embodied, primarily, in their reactive control layer (see Chapter 4) and attention focussing modules (see Chapters 5 and 6) — which gives them the ability to direct their choice of actions dynamically in response to unanticipated real-time events. While such a strategy can virtually guarantee timely responses to a number of localised hard real-time

[13] *Aperiodic* events are those whose start and end times occur at irregular points in time; in other words, asynchronously with respect to the agent's internal operations.

events, it also means that TouringMachines' precise action sequences and long-term goal behaviours can no longer be pre-determined with absolute certainty.[14] Then again, since TouringMachines are resource-bounded systems operating in an unpredictable world, this is only to be expected: as argued above, just as the correctness of TouringMachine's behaviours might need to be traded-off against their robustness, so too might the long-term predictability of their task-level activities.

Although TouringMachines, then, can at best only strive to satisfy their long-term task requirements, they do however possess a number of design features — some already described above, others to be considered in more detail in the next three chapters — which are often viewed as being necessary or at least highly desirable for any embedded real-time system to operate successfully in complex environments. These include:

- continuous monitoring of the outside world to provide the agent with immediate feedback on any dynamic events that might be taking place;

- guaranteed constant-bounded operational latency within and between all of the agent's main functional and behavioural modules to ensure a minimum level of responsiveness in the agent as a whole — these modules include the Perception Subsystem, layer $\mathcal{R}$, layer $\mathcal{P}$, layer $\mathcal{M}$, and the Action Subsystem;[15]

- context-dependent, priority-based scheduling of action execution events — this is implicit in the domain-specific programming of the agent's mediating (suppressor) control rule set;[16]

- an embedded reactive control component (layer $\mathcal{R}$) which ensures behavioural robustness (by virtue of operating in parallel with the agent's slower, non-real-time deliberative components) [Kae87, page 399], and

[14]For example, since a TouringMachine can only travel so fast, it is quite possible that in a situation where the agent's progress is impeded long enough by some outside force (for example, another agent), it might end up being unable to meet its original navigational task deadline.

[15]This suggests that the TouringMachine architecture satisfies Kaelbling's definition of a real-time system: namely, one which guarantees a constant bound on the length of time between the system receiving a specific input and the system generating a response which might have depended on that input [Kae87, pages 399–400].

[16]In the parlance of real-time systems [SR88], a TouringMachine's mediatory control rules can viewed as implementing a scheduler which is *static* (the control rules are fixed throughout the agent's operation), *online* (the rules deal with processes as they occur in the world, including aperiodic ones), *preemptive* (action commands of high priority are able to preempt those of lower priority), and *opportunistic* (the agent's resulting activity occurs through responding to dynamic events in real time).

which guarantees time-critical responses to any of the agent's hard real-time task constraints;[17]

- multi-tasking control — control layers operate concurrently on different aspects of the agent's task set;

- context-dependent selectivity of perceptual input to protect the agent against any possible perceptual overload — this is achieved primarily through the use of heuristic focus of attention mechanisms in layers $\mathcal{P}$ and $\mathcal{M}$;

- a facility for explicit temporal reasoning about long-term task deadlines (the agent's soft real-time constraints) as well as a mechanism for detecting and resolving goal conflicts which can arise as a result of threatened goal deadlines.

Because TouringMachines will almost constantly be in a state of perceptual, cognitive, and action overload, they will generally not be able to perform all potential operations in a timely manner. This, it should be noted, has less to do with precisely how fast TouringMachines can operate than with the fact that they are inherently rationally bounded. Following Hayes-Roth's philosophy, the aim in designing TouringMachines , therefore, is not to create agents which are optimised for the performance of a single pre-determined task; rather, it is to design a control architecture which is capable of satisficing performance over a *range* of tasks, each potentially differing in terms of their required functionality, knowledge sources, and associated time constraints [HR90]. In the TouringMachine framework, then, achieving real-time performance should really be viewed as *one of many* of the agent's objectives, which it will be able to achieve to a greater or lesser extent depending on prevailing environmental conditions and the availability of necessary resources. Before considering these issues further in Chapter 8, more detailed descriptions of the three TouringMachine control layers will now follow.

[17] In fact, in sufficiently time-constrained, fast-paced, or densely cluttered environments, it might still be possible for an agent to miss a hard deadline and perhaps collide. This, as elaborated further in Chapters 4 and 8, results from a TouringMachine's control layers requiring at least a minimum constant amount of time to process and respond to any given external stimulus.

4

TouringMachines – Reactive Layer

Those who will not reason
Perish in the act:
Those who will not act
Perish for that reason.

W.H. Auden

4.1 Introduction

To operate in complex domains, an autonomous agent must be capable of executing timely responses to unexpected or critical events. In realistic domains, the agent will not always have sufficient time or computational resources to choose the *best* action for each situation it might find itself in. Thus, to ensure that the agent at least take an *appropriate* action, particularly in situations where detailed planning or decision-making can be regarded as too expensive, it is vital to provide the agent with a suitable non-deliberative or *reactive* capability.

Critical events faced by an agent are those which either threaten the agent's liveliness or which simply must be responded to within strict, task-specific time limits. For example, in the TouringWorld domain, the sudden appearance of an obstacle within the agent's sensing range would constitute a critical event requiring immediate attention. In other words, the reactive layer is responsible for dealing in a timely and appropriate manner with the agent's *hard* deadlines — those which it cannot afford to ignore or miss under any (reasonable) circumstances.

A TouringMachine's reactive capability is embodied in a control module, $\mathcal{R}$, which is independently connected to the agent's sensors and effectors, the main component of which is a set of domain-specific *situation-action* rules. To guarantee a suitable degree of reactivity, this layer uses neither search nor inference when determining which rule to select at any given time. At this level of description, no explicit model of the world is employed — only the "here and now" of the agent is of concern. Thus, although these rules can make references to the agent's current physical state (see below), layer $\mathcal{R}$ is essentially "myopic" in that it neither holds nor infers any knowledge about the consequences of the actions it proposes.

Although a TouringMachine's reactive capability is somewhat inflexible (the agent's reactions are, after all, "hardwired" to its situational input), layer $\mathcal{R}$ nevertheless provides the agent with a high degree of robustness by minimising the amount of time needed to determine which action should be proposed. Through its situation-action rules, layer $\mathcal{R}$ permits a close coupling of perception to action as well as a reasonable level of dynamic interaction between the agent and its environment. At this level of operation, then, a TouringMachine can be considered *situated* in its environment — an attribute widely considered not only desirable but also critical for successful operation in dynamic, multi-agent environments [AC87, Kae87, CGHH89, HC90, Mae90].

4.2 Operation of Layer $\mathcal{R}$

As an independent control layer, $\mathcal{R}$ operates concurrently with layers $\mathcal{P}$ and $\mathcal{M}$. Like these other layers, $\mathcal{R}$ also receives input from the agent's Perception Buffer and sends output to the agent's Action Buffer (see Figure 4.1). Each of these I/O operations is performed in a synchronous fashion: input is received at the beginning of a timeslice and output is generated at the end of each timeslice. Perceptual input in the form of symbolic multi-attribute information records (see Figure 3.3 *(b)*) representing the physical configurations of any perceived entities — including that of the agent itself — is received and passed to the layer's *Situation-action Rule Set* for further consideration.

A situation-action rule consists of two parts: a *condition set* and an *action*. A condition set is composed of one or more conditions, each of which can either be an arithmetic test on the received perceptual input (for example, `speed(Agent1)` $>$ `0`) or a domain-specific, typed predicate (for example, `is-in-front(Agent1, Agent2)`, which is satisfied if any part of the contour of `Agent1`, an entity of type `agent`, falls within the frontal sensing arc of `Agent2`).[1]

[1]The only other predicate used to date has been `is-behind(Agent1, Agent2)` which is

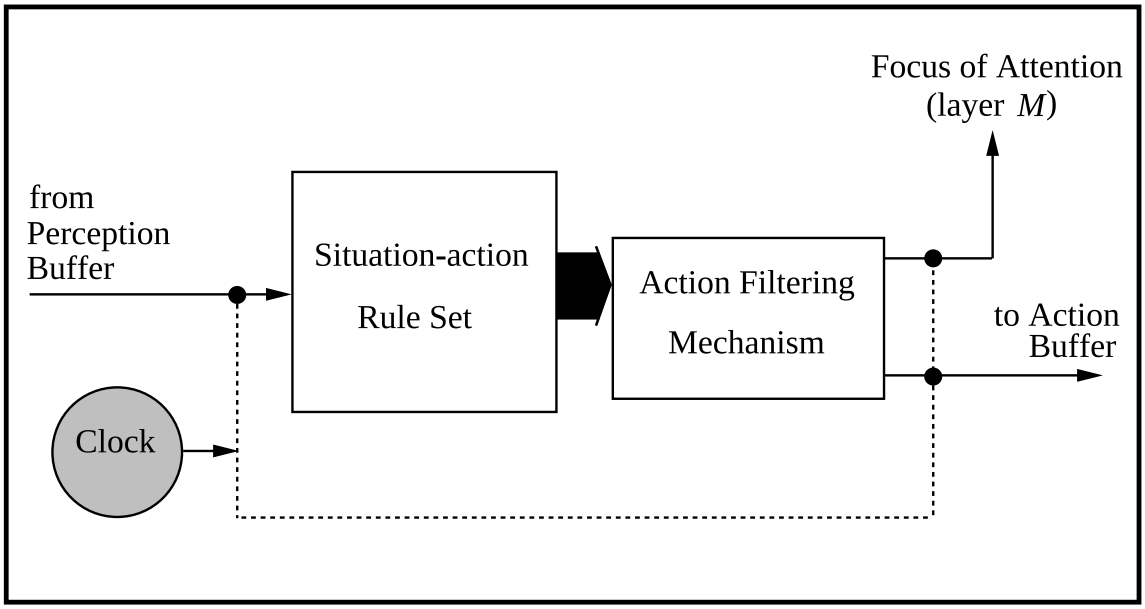

Figure 4.1: Top-level view of the reactive control layer $\mathcal{R}$.

The action part of a situation-action rule corresponds to a reactive command intended as a control sequence for the agent's effectors. In the TouringWorld domain, a reactive command is one of `change-speed` (accelerate by some positive or negative amount) or `change-orientation` (turn the agent's steering wheel through some clockwise or counterclockwise angle).

The rules operate as daemons: each is sensitive to certain environmental stimuli and will independently trigger its action command if its conditions are satisfied by the current perceptual input. Because rules operate in parallel, more than one rule might trigger during a given timeslice. Consequently, an *Action Filtering Mechanism* is used so that at the end of the current timeslice layer $\mathcal{R}$ will be left with at most one action to submit to the agent's Action Buffer (see Figure 4.1). At present, the filtering mechanism employs a single heuristic to determine which rule's action should be chosen, namely, that which was triggered by the environmental stimulus spatially nearest the agent.

In terms of their purpose and level of abstraction, layer $\mathcal{R}$ rules are very similar to Kaelbling's reactive behaviours [Kae87], to reflexes in the Phoenix architecture [CGHH89, HC90], and to the situation-action rules of Pengi [AC87], among others. Like Agre and Chapman's Pengi agent, a Touring-

satisfied if any part of the contour of an agent `Agent1` falls within the rear sensing arc of agent `Agent2`. In addition, some of the reactive rules used in the TouringWorld domain make use of a number of arithmetic and trigonometric functions: for example, `separation(Entity1, Entity2)` which returns the distance between the Cartesian locations of two named entities (see Section 4.3)

Machine's reactive behaviour arises from the interaction between its reactive rules and the agent's current environmental situation. If, for example, a TouringMachine is being controlled by a rule which causes it to flee from an approaching agent, it will only stop doing so when the encroaching agent alters its speed or otherwise falls outside the area within which the TouringMachine's fleeing rule is triggered. After that, the TouringMachine's reactive behaviour will be controlled by whatever other rule gets stimulated into action. In other words, at this level of control, a TouringMachine is not driven by any preconceived idea of what event will occur next. The behaviour of layer $\mathcal{R}$ is opportunistic, and so provides the TouringMachine with a high degree of robustness under uncertainty.

In principle, layer $\mathcal{R}$ is also similar in behaviour to a type of cognitive system Fodor calls an *input process*: a modular, non-intellectual, domain-specific, quasi-reflexive, informationally encapsulated process which may run contrary to what "reasoning" would dictate [Fod83].[2] Such *vertical* processes, as Fodor calls them, buy speed at the price of unintelligence. They contrast with what he calls *central processes* which exploit information from outside of the process, for example memory, in order to make decisions. In this sense, layers $\mathcal{P}$ and $\mathcal{M}$ would be considered central. He also suggests that because input processes compute representations of the layout of environmental stimuli on the basis of less information about the environment than the agent actually has, these computations will need subsequent corrections in light of the agent's background knowledge and in light of the decisions made by other processes. As described in the previous chapter, this is precisely why a TouringMachine's control layers, particularly layer $\mathcal{R}$, require mediation by a global control framework to produce effective behaviour.

As a final operational detail of layer $\mathcal{R}$, it should be mentioned that a second output is generated by layer $\mathcal{R}$ (as shown in Figure 4.1). Specifically, if a situation-action rule triggers, and that action is chosen by the Action Filtering Mechanism, a message is sent by layer $\mathcal{R}$ to the Focus of Attention module in the modelling layer $\mathcal{M}$ informing $\mathcal{M}$ of the type of entity that caused the rule to trigger. The implicit assumption on which this is based is that if the agent has just reacted to some entity it could well be that the agent's modelling layer does not possess a model of this entity: if it had a model it probably could have avoided this near-miss with it. In a limited way, this ensures that the agent's modelling layer is kept up-to-date with certain aspects of the agent's changing surroundings. More detail on agents' focus of attention capabilities will be given in the next two chapters.

[2] A *quasi-reflexive* process is one which is computationally more elaborate than a true reflex. *Encapsulation* relates to the range of information that a process needs to access in deciding what result to produce.

4.3 Reacting in the TouringWorld

To illustrate the range of reactive behaviours available to a TouringMachine operating in the TouringWorld, the complete set of situation-action rules that comprises layer $\mathcal{R}$ is given below in pseudo-code form. Throughout this set, bold terms, for example **KerbThreshold**, are user-defined parameters which are used to configure an agent's reactive layer. These and several other parameters that are used to configure TouringMachines for different TouringWorld scenarios will be described in more detail in Chapter 7 and Appendix A. Seven rules have been used:

```
rule-1:  kerb-avoidance
    if is-in-front(Kerb, Observer) and
       speed(Observer) > 0 and
       separation(Kerb, Observer) < KerbThreshold
    then
       change-orientation(KerbAvoidanceAngle)
```

enables the agent `Observer` to avoid hitting a kerb. Example values in the TouringWorld : **KerbThreshold** = 0.5 metres and **KerbAvoidanceAngle** = $|\pi -$ `orientation(Observer)`$|$ (making the agent's angle of deflection with respect to the given kerb equal in size to its angle of approach).

```
rule-2:  wall-avoidance
    if is-in-front(Wall, Observer) and
       speed(Observer) > 0 and
       separation(Wall, Observer) < WallThreshold
    then
       change-speed(WallAvoidanceSpeed)
```

enables the agent to avoid hitting a wall. Example values: **WallThreshold** = 1 metre and **WallAvoidanceSpeed** = -1 $\times$ `speed(Observer)` (which would bring `Observer` to a full stop short of `Wall`). It should be noted that walls and kerbs are reacted to differently in the TouringWorld merely for the sake of behavioural variety.

```
rule-3:  front-agent-avoidance
    if is-in-front(Other, Observer) and
       speed(Other) < speed(Observer) and
       separation(Other, Observer) < FrontalAgentThreshold
    then
       change-speed(FrontalAvoidanceSpeed)
```

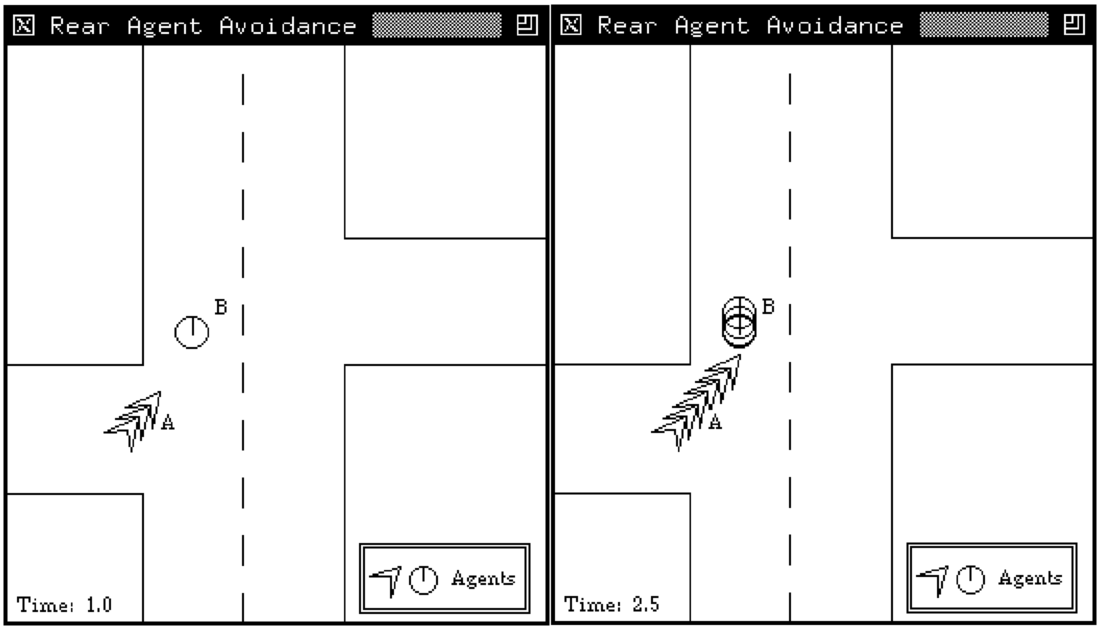

Figure 4.2: An agent will react to another if it is approached too closely. In the left-hand frame, the chevron-shaped agent A starts to encroach upon the stationary round-shaped agent B from B's rear; once A falls within B's collision avoidance range (right-hand frame), agent B starts to pull away to avoid a collision. (In this example agent A has been made effectively non-reactive by setting its **FrontalAgentThreshold** parameter to 0.)

enables the agent to avoid *hitting* another agent `Other` from behind. Example values: **FrontalAgentThreshold** = 2 metres and **FrontalAvoidanceSpeed** = `speed(Other) - speed(Observer)` (which would lower `Observer`'s speed down to that of `Other`).

```
rule-4:  rear-agent-avoidance
    if is-behind(Other, Observer) and
       speed(Other) > speed(Observer) and
       separation(Other, Observer) < RearAgentThreshold
    then
       change-speed(RearAvoidanceSpeed)
```

enables the agent to avoid *being hit* by another agent `Other` from behind. Example values: **RearAgentThreshold** = 2 metres and **RearAvoidance-Speed** = `speed(Other) - speed(Observer)`. The use of this rule is illustrated in Figure 4.2.[3]

[3]It is worth pointing out that all graphical figures like those illustrated in Figures 4.2 and 4.3 which bear the $\mathcal{X}$ Window System† logo and title bar are, in fact, actual outputs generated by the TouringWorld Testbed program. This and other capabilities of the Touring-

```
rule-5:  obstacle-avoidance
    if is-in-front(Obstacle, Observer) and
       speed(Observer) > 0 and
       separation(Obstacle, Observer) < ObstacleThreshold
    then
       change-orientation(ObstacleAvoidanceAngle)
```

enables the agent to avoid hitting an obstacle (anything other than a kerb, wall, or agent). Example values: **ObstacleThreshold** = 1 metre and **ObstacleAvoidanceAngle** = +/- 20°, the latter's sign depending on the size of the obstacle-to-kerb gap either side of `Obstacle`.

```
rule-6:  lane-marking-avoidance
    if is-in-front(LaneMarking, Observer) and
       speed(Observer) > 0 and
       separation(LaneMarking, Observer) < LaneMarkingThreshold
    then
       change-orientation(LaneMarkingAvoidanceAngle)
```

enables the agent to avoid straying over a lane marking. Example values: **LaneMarkingThreshold** = 0.5 metres and **LaneMarkingAvoidanceAngle** = $|\pi -$ `orientation(Observer)`$|$.

```
rule-7:  direction-normalising
    if speed(Observer) > 0 and
       orientation(Observer) ∉ {0°, 90°, 180°, 270°}
    then
       change-orientation(nearest-normal-angle(Observer))
```

enables the agent to change its orientation to the *normal* angle (0°, 90°, 180°, or 270°) nearest its current orientation. Note: this only improves an agent's efficiency because in the TouringWorld domain, path orientations are always orthogonal to these normal angle values (see Section 7.3). The effects of this and several of the above rules are illustrated in Figure 4.3.

The purpose implicit in this set of rules is to prevent the agent from colliding with other world entities. An exceptional rule is `rule-7` whose purpose is almost more aesthetic than practical. More importantly, however, `rule-7` differs from the other rules in that it triggers in the *absence* of environmental stimuli — there is no reference to any other world entity in its condition set. Because it will trigger during most timeslices, it is given less priority by the

World Testbed are described later in Chapter 7. †$\mathcal{X}$ *Window System* is a trademark of the Massachusetts Institute of Technology.

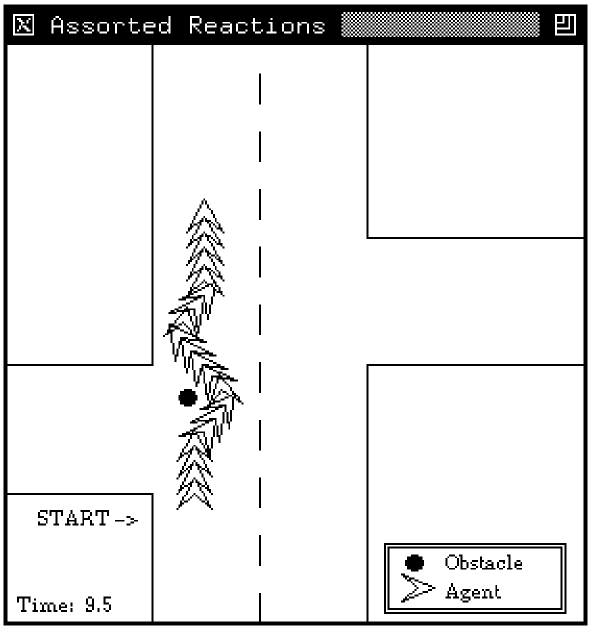

Figure 4.3: Appropriate situation-action rules can enable an agent to avoid obstacles and kerbs, prevent it from straying over lane markings, and, when no other events require attention, adjust the agent's orientation to one of four orthogonal directions.

Action Filtering Mechanism than any other of the above rules. Indeed, it is only ever selected if no other rules trigger during a given timeslice.

Apart from the behaviour to avoid being hit from behind by another agent (`rule-4`), the above rules control the TouringMachine either by reducing its speed or by changing its orientation. By and large, these behaviours have been found to be effective at keeping the agent from colliding with other world entities. However, because situation-action rules maintain no history of past events and make no inferences about future events, they will clearly be of limited use in certain task-domains. For example, reacting to avoid one obstacle might actually drive the agent into another if the two obstacles are close enough together. How useful the particular set of rules described above might be in a given situation depends on many factors, including the agent's speed, the density of obstacles on the agent's path, and the sensitivity thresholds of the agent's rules. These and several other issues are examined in more detail in Chapter 8. Additionally, it is important to remember that layer $\mathcal{R}$ is but one of three layers used in controlling TouringMachines; the usefulness of layer $\mathcal{R}$ can only be fully appreciated by understanding the roles played by TouringMachines' planning and modelling layers. These are the subjects of the next two chapters.

5

TouringMachines – Planning Layer

In the search of truth there are certain questions that are not important. Of what material is the universe constructed? Is the universe eternal? Are there limits or not to the universe? What is the ideal form of organisation for human society? If a man were to postpone his search and practice for Enlightenment until such questions were solved, he would die before he found his path.

Buddha

5.1 Introduction

A sophisticated agent operating in a real-world domain must be capable of accomplishing complex tasks. A task can be considered complex for several reasons. For instance, it might have to be carried out over a long period of time. The task might, in fact, be a composite of several smaller tasks, each requiring the agent's attention at distinct times or spatially-distant locations. Additionally, the task might be incompletely specified or it might specify certain prior restrictions on the way the agent is allowed to tackle it.

One of the oldest problems in the field of AI has been the design of computational systems which, given an initial description of some goal or task, can produce a sequence of actions or *plan*, the successful execution of which will result in the agent having accomplished the assigned task. As past research shows, constructing a sophisticated, general-purpose *planner* which is capable of solving complex tasks is itself non-trivial, requiring the consideration of many deep and complex issues: for example, how to represent and reason about plans and actions, how to control the search of the agent's plan space, or how to generate optimal plans for a collection of simultaneous or conjunctive tasks [Geo90, HTD90].

TouringMachines are primarily task-oriented agents and as such should be capable of generating and executing sequences of actions to achieve the tasks assigned to them. The skills required to do this are many and varied. Among others, these include the ability to decompose large, composite tasks into simpler subtasks; the ability to reason about the plans adopted or actions taken to achieve these tasks; the ability to reason strategically or predictively about the outcome of a plan and about the effects the plan will have on the environment; and the ability to monitor progress of plans and to take appropriate action when the intended effects of these plans are not realised. Furthermore, since TouringMachines are intended for use in realistic domains, they must be capable of carrying out these assigned tasks under real-time pressure, using limited computational resources, and in the presence of other agents and other external events.[1]

In realistic environments, agents will need to handle dynamic events at several levels of granularity. As a result, the decision to implement an agent's necessary operating skills as separate, distributed activity-producing layers has been adopted as a principal tenet of the TouringMachine architecture. With that in mind, the required set of skills for carrying out planned tasks can be usefully divided into two classes: those pertaining to the construction and execution of basic, single-agent plans which handle the initial tasks assigned to the agent; and those addressing what might best be called the agent's *metaplanning* requirements: reasoning about the interactions among different tasks and plans, monitoring the execution of plans and applying techniques to repair failed plans, and reasoning about the interactions among the goals and plans of other agents. It is along this division of planning functions that two of the three layers comprising TouringMachines have been defined: namely, layer $\mathcal{M}$ — which handles the metaplanning skills and will be discussed in the next chapter — and layer $\mathcal{P}$ — which handles the agent's primary planning skills.

[1]It is important to distinguish the notion addressed here of *individual* agents in a multi-agent world generating plans purely for their own use, from the DAI notions of *multi-agent planning* — wherein a single agent generates plans for multiple agents to work on; and *distributed planning* — wherein a single, common plan is produced through the cooperation of multiple agents. The latter two notions are not addressed in this dissertation. See Bond and Gasser [BG88, pages 15 and 21–22] for further details.

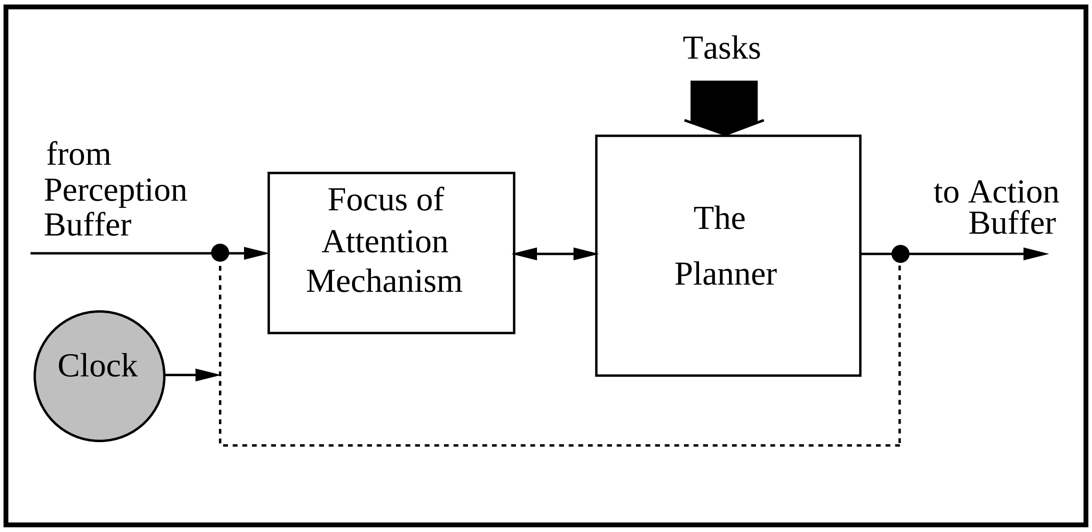

Figure 5.1: Top-level view of the planning control layer $\mathcal{P}$.

5.2 Overview of Layer $\mathcal{P}$

The main objective of layer $\mathcal{P}$ is to build and execute plans which carry out the agent's assigned tasks or goals. More precisely, layer $\mathcal{P}$ deals with one specific class of agent goals called *achievement* goals. An achievement goal is one which has a well-defined set of start and final states in the agent's state space such that, upon arriving in a final state, the goal would be considered achieved and thus able to be terminated. In the TouringWorld domain, for example, navigating to a given location or landmark would be an achievement goal. This class of goals contrasts with the agent's preservation or *homeostatic* goals which are achieved continuously by the agent and have no well-defined final states [CL91]. In TouringMachines these goals are handled by the modelling layer $\mathcal{M}$, so their treatment will be deferred until the next chapter.

Layer $\mathcal{P}$, like layers $\mathcal{R}$ and $\mathcal{M}$, operates as an independent control module. In a synchronous fashion, $\mathcal{P}$ receives input from the agent's Perception Buffer at the start of a timeslice, and sends output to the agent's Action Buffer at the end of the timeslice (see Figure 5.1). In the TouringWorld domain, $\mathcal{P}$'s input takes the form of symbolic multi-attribute information records — see, for example, Figure 3.3 *(b)* — which describe the physical configurations of any perceived world entities, including that of the agent itself; in particular, these records include information about each entity's current Cartesian location, speed, acceleration, orientation, plus any information that the entity might be communicating (for example, if its brake lights are on). Output takes the form of control commands for the agent's effectors and will be discussed in more detail below.

The functionality of this layer, as shown in Figure 5.1, is shared between two major components: the *Focus of Attention Mechanism* and the *Planner*. Unlike layer $\mathcal{R}$ whose operations were sufficiently fast to justify direct manipulation of the agent's perceptual input, layer $\mathcal{P}$'s planning operations, which include potentially lengthy searches of the agent's plan space, are relatively computationally intensive. TouringMachines are intended for use in dynamic worlds, and, in order to guarantee a suitable degree of reactivity from the agent as a whole, the inter-operation latency of any of its planning activities must be bounded. One way to achieve this is to limit the amount of information that layer $\mathcal{P}$ will be required to store and manipulate. This, as described in Section 5.3, is the main purpose of the Focus of Attention Mechanism.

The Planner is charged with taking a high-level description of a given achievement goal or task and constructing and executing a plan for it which, when carried out, will result in the goal being accomplished. As noted above, the functional requirements for an optimal, general-purpose planner are numerous. What seems to be the case is that such *functional* requirements conflict with a TouringMachine's *behavioural* requirements: namely, that given limited computational resources they be capable of operating successfully in complex environments. As Sanborn and Hendler have argued, there is no general way, in dynamic environments, to capture all of the relevant aspects of the domain that may be relevant to the agent's planner [SH88].[2] McDermott has similarly argued that, "in general, it is not worth a special effort to make sure that every foreseeable interaction is forseen, since there will always be some unforeseeable ones that will have to be dealt with." [McD90, page 240]. Goal attainment may not be possible if the agent's environment is sufficiently "hostile". In fact, the only foolproof way to know if an agent's plans are correct is to execute them and find out. What this suggests, then, is that to construct a *practical* planner, one will need to tradeoff between generality and optimality of the planning process on the one side, and flexibility and responsiveness of operation on the other.[3] The "solution" to this tradeoff must come from consideration of the agent's domain of application.

TouringMachines are required to navigate to some assigned location within given time bounds. Intuitively, then, it would make sense for them to perform a certain amount of forward planning — for example, by constructing a route-level path. Indeed, Latombe has argued that even if an agent's initial planned path turns out not to be *the* motion plan used (due, perhaps, to in-built un-

[2] While plan generation in even fairly simple, static domains has been shown to be exponentially hard [Cha90], Sanborn and Hendler have recently shown that the analogous problem in dynamic domains is in fact undecidable [SH88, page 96].

[3] Hendler *et al.* also refer to this as trading precision in decision making for time in responding to events [HTD90, page 71].

certainties in the planner's world model), the plan will nevertheless remain an essential piece of information since it can be used, for instance, to provide the agent with important navigational landmarks for use at execution time [Lat91]. All the same, while some amount of forward planning would seem to be useful, the dynamic and unpredictable nature of the TouringWorld domain would also intuitively suggest that large, detailed plans would be very likely to fail before their successful completion.

Described more fully below in Section 5.4, the resulting TouringMachine Planner should be viewed as a *pragmatic* solution aimed at coping with such task- and domain-imposed constraints. By incorporating such features as hierarchical decomposition, interleaved generation and execution, time-bounded and suspendable operation, and deferred choice of execution method, the Planner enables TouringMachines to behave as purposive, goal-driven agents — despite operating in dynamic environments. Before going into the operational details of the Planner any further, however, a description of layer $\mathcal{P}$'s attention focussing capabilities will prove useful.

5.3 Focus of Attention

TouringMachines are, by definition, computationally resource-bounded. In particular, they have an in-built limit on the amount of available resources for acquiring, representing, or processing information about their surroundings. In other words, TouringMachines have *limited attentional capacity*: by expending computational resources on one particular event, they will necessarily decrease the resources available with which to consider other events [Gre87, pages 59–61]. To ensure, then, that a TouringMachine's planning layer provides timely and appropriate responses under a variety of changing conditions, the agent must be able to reduce the set of all *perceived* world events to the set containing only those events which it considers *relevant*. That is, layer $\mathcal{P}$ needs to control its perceptual intake by focussing attention on important events while ignoring those of lesser importance.

In designing an attention focussing capability many issues need to be addressed. The most important is ensuring that any events which are focussed on remain relevant to the agent's changing context. Also important is to ensure that the mechanism provides the agent with an *appropriate* level of sensitivity vis-à-vis whatever changes are taking place in the world. While the agent will generally only require information about events that are related to its assigned task, not noticing some unexpected event (for example, a nearby agent coming to a sudden halt) could prove catastrophic. Focussing on selected events also implies that some other events will either be ignored

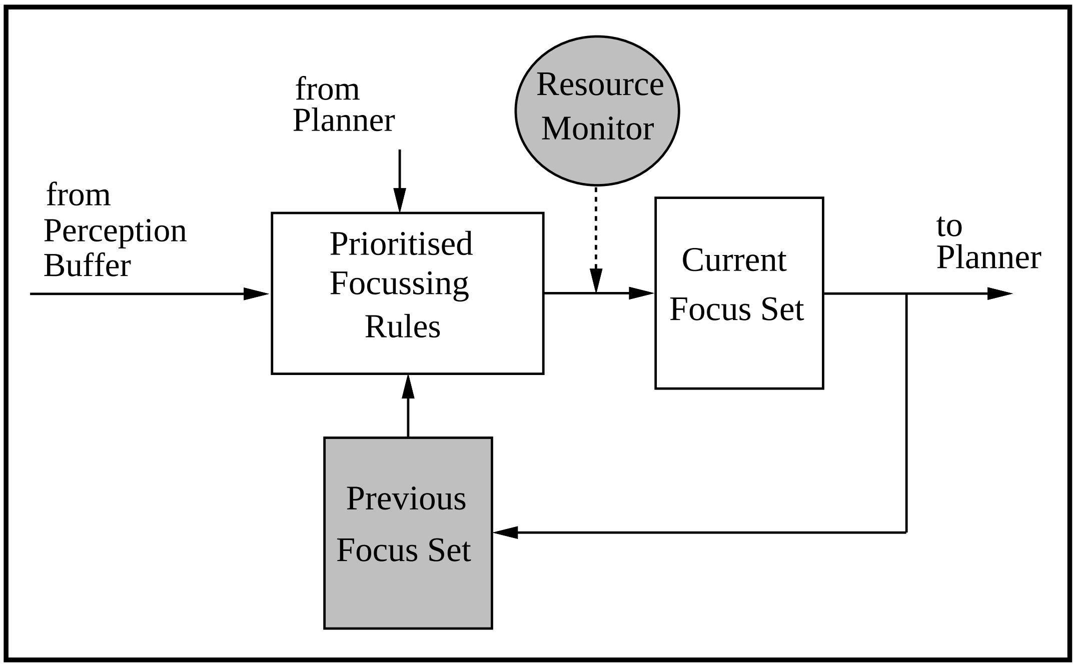

Figure 5.2: The Focus of Attention Mechanism.

outright or not modelled as thoroughly, thereby increasing the agent's degree of uncertainty about its surroundings. Thus, since the agent will occasionally focus on inappropriate events, the issue of how to cope with bad judgements should also be addressed. These matters will be looked at in the next section and also later in Section 6.3.

5.3.1 Operation

The main component of the *Focus of the Attention Mechanism* is a set of user-specified *Prioritised Focussing Rules* which selectively filter information from the agent's Perception Buffer and produce output in the form of a *Current Focus Set* (see Figure 5.2). The Current Focus Set contains perceptual information about only those world entities that were considered "relevant" according to the heuristic selection criteria expressed in the focussing rules. The information contained in the Current Focus Set is then used by the Planner during the remainder of the current timeslice.

Focussing rules are *if-then* rules whose conditional parts are conjunctions of Horn-clause predicates which express domain-specific relations about or between different entities in the agent's world (see Figure 5.3 for the predicates used in the TouringWorld domain). These predicates are always applied from the perspective of the agent doing the focussing and are always relative to the present world time. Thus, for example, if A_1, who is the focussing agent, has perceived at the start of the present timeslice some other agent A_2, then A_2 will satisfy the predicate `is-behind` if A_2 is behind A_1; that is, if any part of the contour of A_2 falls within the circular arc corresponding to the range of A_1's rear sensors. As entities alter their behaviours or move about in time or space, the relationships between the focussing agent and these other entities will change. When relationships change, different focussing rules will be satisfied and so different entities will be focussed upon as time progresses.

The Attention Mechanism possesses a finite number of user-ordered focussing rules and these are applied sequentially to each of the entities appearing in the current perceptual input. Any entity which satisfies each of the conditions of the particular rule being applied are deemed "relevant" and placed in the Current Focus Set. However, since both the number of perceived entities and the number of focussing rules being applied could be potentially large, the total number of entities that get placed in the Current Focus Set has to be controlled if the inter-operation latency of layer $\mathcal{P}$ is to be guaranteed. In the focussing mechanism, this is done by "charging" for each entity that is focussed upon; that is, from an initial pool of resources made available for focussing at the start of each timeslice, deductions are made to the pool's total resources each time an entity is selected for placing in the Current Focus Set. Focussing stops, then, when the available resources run out or when all focussing rules have been considered and no new entities have been selected.[4]

The predicates appearing in focussing rules can be used to express three types of relations over world entities: *physical*, which relate to some aspect of the perceived entity's spatial or physical configuration (for example, `is-behind` or `is-stopped`); *identity*, which relate to an entity's type or identity (for example, `is-an-agent`); and *temporal* (for example, `is-novel`). Temporal predicates differ slightly from the two other types in that, while still relating to events perceived in the present timeslice, they are used in making comparisons between an entity's *current* configuration and that exhibited in the *previous* timeslice. So, for example, an entity will satisfy `is-novel` if it has been perceived in the current timeslice and if there is no record of it having been perceived in the previous timeslice. To handle such predicates a copy of the Current Focus Set is taken at the end of each timeslice and placed in a buffer called the *Previous Focus Set* (see Figure 5.2). The contents of this buffer are then used by the focussing rules during the next timeslice as a historical record of what was focussed on in the immediate past. Several other temporal predicates used in the TouringWorld domain are listed in Figure 5.3.

A TouringMachine's Focus of Attention Mechanism functionally approx-

[4]Total focussing resources and unit costs of focussing on world entities are TouringWorld Testbed parameters which are described more fully in Chapter 7.

Physical relations:
- `is-behind`
- `is-in-front`
- `is-within-distance-D`
- `is-stopped`
- `is-travelling-over-speed-S`
- `is-communicating-signal-C`

Identity relations:
- `is-an-entity`
- `is-an-agent`
- `is-an-object`
- `is-self`

Temporal relations:
- `is-novel`
- `has-disappeared`
- `has-changed-speed`
- `has-changed-direction`
- `has-started-communicating-signal-C`

`D`, `S`, and `C` are distance, speed, and signalling variables respectively.

Figure 5.3: Focussing predicates used in the TouringWorld.

imates the Attention Director of Wood's AUTODRIVE agents [Woo90] as well as the filtering capability of the perception subsystems in Hayes-Roth's blackboard-based architecture [HR90]. It is effective at generating an up-to-date, context-sensitive focus set and is also somewhat robust to errors; because TouringMachines constantly monitor the world and are able to react at all times to unexpected events, temporary focussing errors need not prove fatal. All the same, the TouringMachine mechanism can by no means be considered a complete, general-purpose focussing capability. For instance, the contents of the rule set can only change through explicit requests from the Planner; currently if some part of the agent's plan refers to an entity about which the agent has no information, the Planner will flag the focussing mechanism (see Figure 5.2) to demand that the relevant entity or entity type be focussed on during the next timeslice — the effect of this being that an appropriate focussing rule gets added at the beginning of the ordered rule set. The bulk of the rules, however, are static and need to be specified and ordered by the user in advance.

One way to increase TouringMachines' adaptiveness would be to enable the focussing mechanism to make use, at run-time, of historical information — for example, number and type of rule firings during previous scenarios — to establish an improved statistical ordering of the focussing rules (Russell [RW89] refers to this as *post hoc inductive metalearning*). This capability, however, has not been implemented but would be a possible candidate for future work involving learning (see Chapter 9).

5.4 The Planner

5.4.1 Overview

The Planner generates and executes plans as a means of achieving some initially assigned task of the agent. As described at the beginning of the chapter, layer $\mathcal{P}$, and thus the Planner, is not intended as a general-purpose planning capability. Rather, it is designed to be of some practical use at generating effective goal-oriented behaviours in a multi-agent dynamic world, while at the same time operating as an independent control module in conjunction with two other rather dissimilar ones — layers $\mathcal{R}$ and $\mathcal{M}$.

The Planner exhibits many of the characteristics often considered necessary for practical operation in complex domains. In particular, because lengthy detailed plans are very likely to fail in such worlds, the Planner interleaves plan generation and execution. Unlike many of the early planning systems such as STRIPS [FN90] or NOAH [Sac90], the TouringMachine Planner does not start out with a complete picture of its world. Rather, its knowledge is very much limited to what it currently perceives or has focussed on, and so cannot always be relied upon to produce complete and correct plans for each different situation.

The approach to interleaving adopted here is similar to that used in NASL [McD90] and to the method of *lazy skeletal expansion* used in Phoenix agents [CGHH89, HC90]; that is, plan generation and execution are mixed by choosing one step of a plan at a time and then executing it. The advantage this has is that it makes the Planner less susceptible to unexpected events in the environment and so can potentially help minimise wasted planning effort. On the other hand, as Hendler *et al.* [HTD90] point out, this approach can make it more susceptible to (subplan) interaction errors because it might commit prematurely to particular subplan temporal orderings which later prove incorrect. To alleviate this problem, the Planner has been designed to defer committing to specific choices of plan execution methods until absolutely necessary. Also, since TouringMachines are able, through reaction in $\mathcal{R}$ and execution monitoring in $\mathcal{M}$ both to keep abreast of changes in the world and to take corrective actions when necessary, they should be reasonably well prepared to deal with most eventualities that arise from sub-optimal planning.

It is also widely recognised that in complex domains, planning must be done *hierarchically* at different levels of abstraction to allow the planner to manipulate a simpler but computationally more tractable theory of its world [Wil86]. Abstraction levels are distinguished by the granularity of the discriminations they make of the world. The TouringMachine Planner employs such levels, both in the plan generation process (by producing a partially-elaborated hierarchical subgoaling structure) and in its description of the environment. In particular, plans are used at different abstraction levels to describe only those aspects of the world which are pertinent to the current situation. In the TouringWorld domain, for instance, reasoning about such things as absolute world coordinates is done at a level well below that which is concerned with route-level navigation to some target landmark. In a navigation domain such as the TouringWorld , this is particularly useful since it facilitates the planning of tasks for which full details are not initially available but which can subsequently be obtained during execution of the task (for example, the precise Cartesian location of a given landmark). A similar view has been taken in the design of other navigation-oriented planners such as ELMER [MRS82] and AUTODRIVE [Woo90].

As an independent, synchronously-timed control layer, $\mathcal{P}$'s inter-operation latency must be guaranteed. Since the computational demands of planning would seem, at first glance, to be incompatible with this requirement, certain design constraints must be imposed on the planning process. As a result, the Planner has been designed to work *incrementally* (as Kaelbling describes): doing a few computation steps during each timeslice, then storing its state until the next timeslice [Kae87]. Thus, when the Planner has a real action to effect on the world it will simply execute it; at any other time, the Planner's output will be interpreted as if saying that the Planner does not yet have a complete solution. The advantage of such an *embedded* design, Kaelbling points out, is that it allows the agent's programmer to specify a hierarchical action map which separates — and thus enables concurrent execution of — actions specified in terms of Planner-level operators and actions which are more easily expressed as reflexive reactions [Kae91]. This, of course, is precisely the relationship that exists between layers $\mathcal{P}$ and $\mathcal{R}$.

Since one of the tenets of the TouringMachine architecture is to minimise the amount of work any one control layer has to perform, it is worth under-

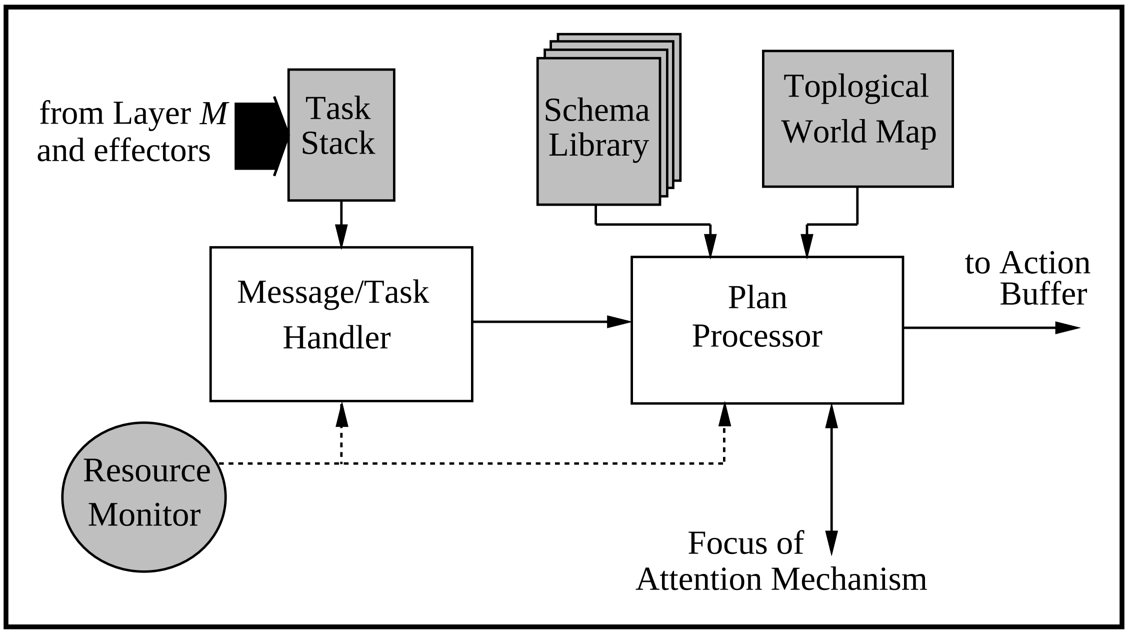

Figure 5.4: The Planner.

lining that the Planner's design has been simplified to act as a single-agent device; in other words, it is designed to act "blindly" as if no other agents were present in its world. The Planner, in effect, assumes the existence of an idealised, exception-free environment and so does have to be burdened with such functions as temporal projection or plan execution monitoring. These, and several other such metaplanning capabilities, are performed by the agent's modelling layer $\mathcal{M}$ and are described in the next chapter. Before considering these, however, some operational details of the Planner will be given first.

5.4.2 Operation

The TouringMachine Planner comprises two main components: a *Message/Task Handler* and a *Plan Processor* (see Figure 5.4). The Message/Task Handler is charged with handling messages requesting the Planner to work on some task. Task requests are received both from the agent's modelling layer and from its effectors. From the former it receives the agent's initial task (for example, to plan some route) as well as any extra tasks that are dynamically generated to deal with certain exceptional events (for example, stopping to give way to another agent); from the effectors it receives requests to iterate certain actions that require more than one timeslice to execute (see **Plan Execution** below). New task requests can be taken on at the start of each timeslice so the Planner's activities can generally be relied upon to reflect the agent's changing task-related needs.

The Plan Processor constructs plans for any task requests passed on to it by the Message/Task Handler. Plans are built by retrieving suitable partially-elaborated plan templates or *schemata* from the *Schema Library* which, upon retrieval, are combined by the Plan Processor to form a *plan structure* for subsequent processing. Similar in function to skeletal plans in Phoenix [HC90] and to procedural KAs in PRS [GLS87], schemata are general representations of some task-related activity to be performed by the Plan Processor. Such activities can either involve further library retrievals to decompose and process a schema's constituent plan sub-steps or *body* (see **Plan Generation** below) or it can involve the execution of some low-level action.[5] As the bodies of schemata are decomposed, their constituent sub-steps are incrementally placed at appropriate points in the Plan Processor's plan structure — a hierarchical, temporally constrained tree representing the Planner's working solution to the given task. Eventually, due to their position in this temporally constrained structure, low-level actions will get selected for processing and subsequent submission to the agent's effectors. The Plan Processor's operation terminates only when the entire generated plan structure has been fully processed and when no further task requests are received from the Planner's Message/Task Handler.

Just as the Focus of Attention Mechanism's operational costs were monitored to guarantee an upper-bound on its computational latency, so too are the Planner's. In particular, the costs of such operations as handling task request messages, retrieving schemata from the library and placing them in the plan structure, as well as executing any low-level actions present in the plan structure, are continuously tallied and deducted from the Planner's per-timeslice resource allowance. When resources eventually run out, the Planner suspends all plan processing and becomes inactive until the beginning of the next timeslice. Such incremental behaviour guarantees the time-bounded responsiveness required of the Planner, in particular, and therefore of layer $\mathcal{P}$ as a whole.

Besides making use of stored plan templates residing in the Schema Library, the Planner can access information from two other sources: the Focus of Attention Mechanism and the *Topological World Map* (see Figure 5.4). Since many of the retrieved schemata make reference to such things as the agent's own location or some other entity's speed, the Planner uses the Current Focus

[5] In fact, just before its operation is suspended at the end of each timeslice, the Plan Processor performs a third activity: it instructs layer $\mathcal{P}$'s Message/Task Handler to send layer $\mathcal{M}$ a message, the content of which specifies the name of the current plan schema being processed. As described in Chapter 6, this information is used by layer $\mathcal{M}$ to maintain a model of the agent's own current intentions.

Set produced by layer $\mathcal{P}$'s Focus of Attention Mechanism to "fill in" appropriate gaps — uninstantiated variables — in the generated plan structure (see below). The Planner also has access to a database which it uses as a long-term, static storage medium; in the TouringWorld domain this database embodies a Topological World Map containing the names and locations of the various paths and path junctions that can be found in the agent's world. This information can likewise be used to instantiate unbound variables in the generated plan structure.

5.4.2.1 Plan Generation

Plan generation starts with the retrieval of an appropriate plan template or schema matching the agent's initially assigned task. Tasks, received as messages from layer $\mathcal{M}$, are represented as *(task name, argument list)* pairs. In the TouringWorld domain, for example, the initial task description

```
(plan-a-route, [[agent, agent1],
                [final-destination, tearoom],
                [travel-time, 30],
                [within-distance, 5]])
```

would be interpreted by the Planner of TouringMachine `agent1` as an instruction to generate and execute a route to some location called `tearoom`, to take no more than 30 units of time getting there, and to end up no more than 5 units of distance off the final target.

Schemata are frame-like structures which are defined by a set of *(**attribute**, `value`)* pairs, as shown in Figure 5.5. Schema retrieval is performed by searching the Schema Library for any schema whose **name** and **arguments** attribute values match the name and argument list, respectively, of the Planner's assigned task.[6] When a suitable match is found the chosen schema is selected and its variables unified with corresponding values from the assigned task description. For example, given the initial `plan-a-route` task from above and the corresponding schema from Figure 5.5, `Destination` would be replaced with `tearoom`. Once instantiated, the schema is used to start the construction of the Plan Processor's hierarchical plan structure: the first instantiated schema — typically `plan-a-route` in the TouringWorld domain — is made the root node of the structure, this root node becomes the

[6] Name and argument matching is done using standard Prolog unification [CM81]. Schema entries starting with a lower case letter (for example, `final-destination` in Figure 5.5) are treated as ground terms and entries starting with an upper case letter (for example, `Destination`) are treated as variables. "_" denotes the anonymous or don't-care variable.

```
name:  plan-a-route
type:  composite
version:  1
cost:  3
arguments:  [agent, Agent],
            [final-destination, Destination],
            [travel-time, TravelTime],
            [within-distance, _]
applicability conditions:  [time1par, lt, time2par],
                           [time2par, lt, time3par]
preconditions:  []
body:  [get-route, [
          [time1par, _],
          [agent, Agent],
          [final-destination, Destination],
          [xagent, XAgent],
          [yagent, YAgent],
          [route, Route],
          [route-length, RouteLength]]],
       [get-route-speed, [
          [time2par, _],
          [agent, Agent],
          [xagent, XAgent],
          [yagent, YAgent],
          [route, Route],
          [route-length, RouteLength],
          [travel-time, TravelTime],
          [travel-speed, TravelSpeed]]],
       [follow-route, [
          [time3par, _],
          [agent, Agent],
          [route, Route],
          [travel-speed, TravelSpeed]]]
postconditions:  []
```

Figure 5.5: A composite schema: top-most schema used in generating a route to some target destination within a given time.

Plan Processor's *current plan*, and this current plan or instantiated schema is then considered for subsequent processing.

The Plan Processor processes instantiated schemata — in other words, plan nodes in its plan structure — according to their **type**. The type of a schema primarily dictates how the Plan Processor is to interpret its executable core or **body**. There are two schema types: *composite* and *primitive*. A composite schema (see Figure 5.5) is one whose body slot contains a list of one or more *body steps*. Body steps are interpreted as subtasks of the initial schema and thus processing a composite schema will result in the Plan Processor performing further retrievals from the Schema Library. A primitive schema (see Figure 5.6) is one whose body contains an executable routine or code fragment which, rather than triggering further Library searches, either performs a numerical computation whose result is to be used elsewhere in the agent's plan structure (for example, the agent's estimated stopping distance) or effects some low-level action on the world (see **Plan Execution** below).

Processing a composite schema amounts, primarily, to decomposing its body of subtasks, retrieving suitable library schemata for these subtasks, and then placing such instantiated child schemata or plans at appropriate points in the Plan Processor's plan structure. To decide where in this structure to place these child plans, the original (composite) parent schema's **applicability conditions** must be considered. The applicability conditions of a schema are a set of *constraints* which are used to establish a temporal ordering for the schema's own body steps. Constraints are triples of the form:

$$[StartTime_1, RelOrder, StartTime_2]$$

each of which expresses a particular temporal or relational ordering, $RelOrder$, between the processing start times, $StartTime_1$ and $StartTime_2$, of a given pair of body steps appearing in the schema's body slot. So, for example, in the `plan-a-route` schema of Figure 5.5, the constraint `[time1par, lt, time2par]` is used to denote the fact that the body step `get-route`, whose first argument is `[time1par, _]`, should be processed before the body step `get-route-speed`, whose first argument is `[time2par, _]`. Similarly, the constraint `[time2par, lt, time3par]` indicates that the body step `get-route-speed` should be processed before `follow-route`. The relational ordering operator `lt` is interpreted by the Plan Processor as "less than": in other words, occurring earlier in time. Other ordering operators that can be used include `le` (less than or equal), `gt` (greater than), and `ge` (greater than or equal).

After comparing all pairs of ordered starting times, the body step corresponding to the earliest starting time is determined and its corresponding schema selected from the library and placed in the Plan Processor's plan structure as the left-most child of the current (parent) node. This is then repeated

```
name: calculate-stopping-distance
type: primitive
version: 1
cost: 5
arguments: [[_, Speed],
             [_, Rate],
             [_, Period]],
            [[stopping-distance, Distance]]
applicability conditions: []
preconditions: []
body: [Temp = Speed * Period + (Rate * Period * Period) / 2,
        truncate(location, Temp, Distance)]
postconditions: []
```

Figure 5.6: A primitive schema: calculates the stopping distance for an agent travelling at a given speed and decelerating at a given rate per unit of time.

for every other body step of the original parent schema, with each new child schema being located to the right of any sibling nodes already placed in the plan structure. Having established a complete ordering for the original schema's body steps (or children) the Plan Processor proceeds with processing the next schema which will be the first (left-most) of these recently placed child schemata. Processing of this schema is then carried out in the same manner as the parent schema. Schemata, thus, are processed in a strict earliest-first postorder (children before parents) manner.

So far, schema processing has been described solely in terms of schema body processing; however, two secondary schema processing activities have yet to be addressed. The first of these, in fact, takes place *before* a schema's body is processed and involves processing of the schema's **preconditions**. The second of these activities takes place immediately *after* the body is processed, and involves processing of the schema's **postconditions**. Essentially, a schema's precondition slot is used to specify conditions about the agent's external world which must be true before processing of the schema's body can commence. The postconditions slot, on the other hand, is used to specify external world conditions which must be true before the Planner can complete processing of the current schema and move on to process the next schema in the plan structure. In the TouringWorld domain, for example, the schema `poll-traffic-light`, which is selected for processing if the agent comes up to a red traffic light, has as its postconditions slot a test to check whether the traffic light in question has changed its colour to green. If the conditions appearing in the preconditions or postconditions of a schema do not match those that are currently true of the world, the Plan Processor suspends all activity and the Planner is exited until the next timeslice, at which point the same schema will reconsidered again. So, for example, the `poll-traffic-light` schema will be processed repeatedly from one timeslice to another until the appropriate traffic light turns green.[7] Unlike applicability conditions, then, which are used to state conditions about events *internal* to the Planner (such as the order in which different subplans should be processed) preconditions and postconditions are used to state conditions about events which are *outside* the Planner's direct control and which may or may not ever take place. While it is conceivable, then, that the Planner could get stuck processing the same schema indefinitely (for example, a particular traffic light never turns green), such an occurrence is considered exceptional and is therefore handled elsewhere in the TouringMachine; namely in layer $\mathcal{M}$.

Before describing the execution phase of plan processing, one final aspect concerning schema retrieval requires addressing. If the agent's Planner can achieve an assigned task in more than one way, its Schema Library will contain a corresponding number of schema templates, each with the same name as the particular task but each distinguished by a unique identifier or **version** number. If the library is being searched for a schema which turns out to have several matching versions, the Plan Processor will decide, based on whether the schema's type is composite or primitive, how many or which ones it should select. In particular, if the schema being retrieved is composite, only one is retrieved, the selected version number in fact being arbitrary. This reflects the view, similar to that adopted in McDermott's NASL planner, that if plan execution and generation are being interleaved, maintaining backtracking points for all possible ways of achieving the planned task — as done in STRIPS [FN90] for example — is simply unsuitable. As McDermott puts it, "if [the plan] fails, the state of the world will in general have been changed enough by the attempt so that the alternative plans are out of date" [McD90, page 227]. If the schema is primitive, on the other hand, again only one schema is retrieved, but in this case all alternative schema version numbers *are* considered as potential backtracking points and are recorded — for possible future use —

[7] Note that the current schema will also be suspended if the entity referred to in its preconditions or postconditions slot is not part of the agent's Current Focus Set. However, as mentioned above in Section 5.3.1, a side-effect which occurs when the Planner makes a reference to a non-focussed entity is that a flag gets sent to layer $\mathcal{P}$'s Focus of Attention Mechanism instructing it to focus on this entity in the future. It is quite likely, then, that the entity will become part of the Current Focus Set by the next timeslice. Of course, if the entity is not presently within sensing range, the current schema will have to remain as the current schema — and be repeatedly processed and suspended — at least until the entity is sighted.

in the plan structure alongside the retrieved schema. This approach reflects the view that while exhaustive backtracking would typically be unsuitable, the ability to defer commitment to a particular primitive schema — that is, to a particular execution method — can potentially help to minimise the extra work that would be required if the Planner's "preferred" execution method later failed for some reason. This will be discussed in more detail shortly.

5.4.2.2 Plan Execution

Repeating the decomposition/retrieval/placement procedure described thus far results in the creation of a temporally constrained, hierarchical action ordering or plan structure. Starting with the original task's retrieved schema as the *root* of this structure, this generative procedure continues until the next schema to process is a primitive schema. Composite schemata, in effect, can be considered *nodes* in this plan structure, while primitive schemata can be regarded as *leaves*.[8]

Primitive schemata, unlike composites, do not have decomposable bodies. Rather, their bodies consist of a routine or code fragment which can be directly executed by the Plan Processor.[9] Two types of operations can be performed in a primitive schema body. The first of these involves computing values for unbound argument variables that are referred to elsewhere in the plan structure. For example, `get-route-speed`, which appears as a body step of the composite schema `plan-a-route` in Figure 5.5, is a primitive schema. It computes the agent's ideal average travelling speed and, through its argument `[travel-speed, TravelSpeed]`, shares its computed result with any other nodes in the generated plan structure which also have `[travel-speed, TravelSpeed]` as an argument (for example, its sibling node `follow-route`). The arguments of primitive schemata are divided logically into input and output arguments (composite schemata only have input arguments). Computations are shared with other schemata in the plan structure by propagating any computed results through the appropriate output arguments. For example, the schema `calculate-stopping-distance` of Figure 5.6 shares its result through its one output argument `[stopping-distance, Distance]`. Results are propagated first to the schema's siblings, and then up through the structure repeatedly to any of the schema's ancestors possessing the appropriate argument.

The second type of operation a primitive schema can perform is that of

[8]More precisely, composite schemata should be considered *AND-nodes* since, as described above, no *choicepoints* — that is, no backtracking points with alternative versions of the schemata — are recorded for these.

[9]Code fragments are written in SICStus Prolog (see, for example, Figure 5.6).

submitting some action command to the agent's Action Buffer. In the TouringWorld domain the set of action commands that the Planner can submit include the same two performed by layer $\mathcal{R}$ (`change-speed` and `change-orientation`) and also a variety of communicative action plans such as `signal-left`, `flash-headlights`, or `honk-horn`. As mentioned previously, the Planner executes one action at a time. More precisely, it submits at most one action command to the agent's Action Buffer per timeslice, after which it suspends its operation until the next timeslice.

Of course, some of the actions submitted by the Planner to the effectors may take several timeslices to execute in full. For instance, if the Planner instructs the agent's effectors to execute a change in orientation of, say, 180° but the agent is only capable of turning 45° per timeslice, it would take four timeslices for this particular action to complete. Rather than permit any single TouringMachine control layer to tie up the effectors over arbitrarily long periods of time, effectors have been programmed to execute as much as they possibly can in one timeslice (a 45° turn, say) and then resubmit the remainder of the action as a future task for the Planner. At the beginning of the next timeslice the Planner's Message/Task Handler will pass the new task to the Plan Processor for processing: the corresponding action will be partially executed again, and the procedure repeated until the action is ultimately completed.[10]

The TouringMachine Planner is resource-bounded and since executing actions must realistically use up resources of some sort, schemata have been provided with a **cost** attribute (see Figures 5.5 and 5.6) which is used to inform the Planner of the minimum number of per-timeslice resources it must have *before* a particular schema can be processed. Whereas the cost associated with a composite schema is intended to cover such operations as its retrieval and subsequent subplan decomposition (and so is calculated as a factor of the number of body steps present in the schema), the cost of a primitive schema is intended to cover the expense of processing its executable body.

The cost attribute of a schema is used in deciding whether to suspend the Planner's operations: if insufficient resources exist either to retrieve a schema from the library or process a schema's body, the Planner's operations are suspended until the next timeslice when its resource pool will be reinitialised. More precisely, if the current schema being processed is composite and resources become sufficiently depleted, the Planner's suspension takes place immediately. If, however, the current schema is primitive, the Plan Pro-

[10]This, of course, is what happens to iterated actions in non-exceptional circumstances. Like any actions submitted by any of the control layers, iterated actions are potential candidates for suppression (see Section 3.3.2) and thus may be terminated prematurely.

```
test preconditions of current plan
if preconditions not satisfied
then {suspend current plan and goto Planner exit}
if current plan is active begin
   if body steps to expand > 1
   then {retrieve/ place schema for selected body step
            and goto process current plan}
   else {retrieve/ place schema for this last body step,
            determine next plan, and goto process current plan}
end else begin
   test postconditions of current plan
   if postconditions satisfied
   then {dequeue current plan, determine next plan,
            and goto process current plan}
   else {suspend current plan and goto Planner exit}
end
```

Figure 5.7: Algorithm to process a composite schema (pseudocode). The *current plan* is the schema in the Plan Processor's plan structure currently being processed. A plan is *active* if it was not suspended in the previous timeslice. The statement "**goto** *process current plan*" is an iterative call to process the current plan (using this algorithm or the one shown in Figure 5.8 depending on its type).

cessor will first consider whether any of the schema's recorded choicepoints (alternative versions of the current schema) have a suitably low cost attribute to warrant further processing during the current timeslice. The idea behind having different versions of the same primitive schema is the same as that of having access to several execution methods with differing computational requirements which solve a common task (for example, different distance ranging schemata with varying degrees of accuracy). At present Touring-Machines are heuristically programmed to maximise per-timeslice resource utilisation and so consider different schema versions in order of *decreasing* cost. If a suitable schema can be found, any remaining choicepoints are discarded and the chosen schema executed; otherwise processing is suspended and then continued in the next timeslice.

Besides executable bodies, primitive schemata, like their composite counterparts, also possess preconditions (which are processed before the schema's body) and postconditions (which are processed immediately after the body). Once a primitive schema has been fully processed it is removed from the plan

```
test preconditions of current plan
if preconditions not satisfied
then {suspend current plan and goto Planner exit}
if current plan is active begin
   if sufficient resources to execute current plan
   then {execute body, propagate appropriate
            variable bindings, dequeue current plan,
            determine next plan, and goto process current plan}
   else if suitable choicepoint exists
   then {retrieve/place schema for selected choicepoint,
            make it the current plan,
            and goto process current plan}
   else {suspend current plan and goto Planner exit}
end else begin
   test postconditions of current plan
   if postconditions satisfied
   then {dequeue current plan, determine next plan,
           and goto process current plan}
   else {suspend current plan and goto Planner exit}
end
```

Figure 5.8: Algorithm to process a primitive schema (pseudocode).

structure. The Plan Processor must then decide which node of the plan structure to process next. The strategy, as described above, is to process first all siblings of the recently removed schema, each time selecting the earliest: in other words, via a postorder search of the plan structure. Once all of these sibling schemata have been processed, the Plan Processor moves up one level in the plan structure, removes the parent of the last schema to be processed, and then commences to process each of the siblings of this latest one. This process is repeated until the root of the plan structure — the schema corresponding to the Planner's original task — becomes the current plan, at which point the Planner suspends all activities.[11] The full pseudocode algorithms

[11]Processing all of the child goals of some parent node before considering any of the parent's siblings implies that interactions are assumed not to occur among the different goals and subgoals in the plan. In other words, the Planner makes the *linearity assumption* [HTD90]. In the TouringWorld task-domain where the solutions to different subgoals (for example, separate stages of the route) can be fairly easily decoupled, this seems to be a reasonable simplifying assumption to make. Clearly though, this decision would have to be reconsidered if TouringMachines were to be given tasks which are less suited to linear planning methods.

for processing schemata are given in Figures 5.7 (composite schemata) and 5.8 (primitive schemata).

5.5 Planning in the TouringWorld

A TouringMachine's Planner is charged with building an initial single-agent plan which, when executed, will accomplish the agent's main goal of reaching some target destination within some given spatio-temporal bounds. Through the use of a Library of template schemata and a Topological World Map containing the locations of all known paths and path junctions,[12] the Planner generates a shortest length route to the target location as well as computing a suitable cruising speed with which to satisfy the given deadline.[13]

After generating an initial ordered list of component path names, the Planner proceeds to tackle in sequence each intermediate stage of the route. To handle individual paths, the Planner has to generate further plans to help determine, among other things, where the agent currently is and the direction in which it has to head to reach the next path in the route. Ultimately, the Planner also has to submit the necessary actions which will physically move the agent in the right direction at the right speed along each intermediate path comprising the route (see Figure 5.9).

At the level of individual paths, the Planner makes use of external landmarks — for example, particular objects or information signs — to establish when certain actions need to be taken. Landmarks, which are listed in the Topological World Map, are associated with different paths or path junctions and can be used, for example, to mark the transition from one path to another (note, however, that these are not visible in Figure 5.9). Since the exact locations of external landmarks are not given in the topological map, the Planner must explicitly look out for particular objects or signs if it is to make all of the correct path-to-path transitions. Although processing sensory input creates extra work for the Planner (landmarks are explicitly tested for in schemas' preconditions and postconditions slots), closely coupling the Planner to the agent's sensors has the advantage of making it less dependent on detailed prior information, and thus potentially less brittle at execution time.

As a single-agent device, the Planner does not usually concern itself with

[12]See Appendix A (page 191) for a syntactical description of an agent's Topological World Map, and Appendix A (page 194) for the complete list of TouringWorld schema template names.

[13]The algorithm used to calculate the shortest length route is based on Sterling and Shapiro's best-first search algorithm [SS86, pages 292–294], appropriately modified to favour routes with the fewest hops; in this case, junction turnings.

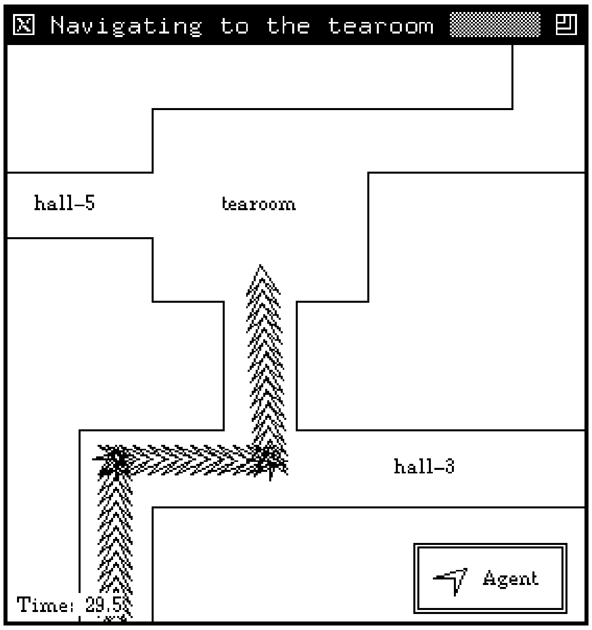

Figure 5.9: In a single-agent, exception-free environment (no obstacles or obstructions of any sort), a TouringMachine's initially generated route plan will be sufficient to accomplish its initially assigned navigation task.

other agents' activities or whereabouts. For example, since the Planner has no prior knowledge regarding the whereabouts of traffic lights or stop signs (these are not recorded in the Topological World Map), TouringMachines only make planned stops when changing direction — that is when leaving one path and joining another. However, since the Planner is intended to operate in complex environments, it should be able to cope with a degree of world change; that is, it should also be capable of taking on new tasks prompted by the agent's changing context. As mentioned above, the Planner takes on extra tasks when appropriate task request messages are received from the agent's modelling layer $\mathcal{M}$. New task requests (for example, yielding to another agent at a junction or stopping at a red traffic light) are considered to be of high priority in the TouringWorld domain and therefore always placed earliest in the Planner's plan structure. Whether the schema retrieved for the new task is composite or primitive, it will be handled accordingly — as if it were simply part of the agent's original task. But, whereas the "mechanics" to *process* dynamically generated tasks is, as this chapter should hopefully have demonstrated, essentially quite straightforward, the steps required to *generate* such tasks are rather more involved and so will be the topic of the next chapter.

6

TouringMachines – Modelling Layer

I sat in my room and I drew up a plan
but plans can fall through as so often they do
and time is against me now...

Morrissey, *Accept Yourself*

6.1 Introduction

Like most real-world domains, the TouringWorld is populated by multiple autonomous entities and so will often involve dynamic processes which are beyond the control of any one particular agent. For a planner — and, more generally, for an agent — to be useful in such domains, a number of special skills are likely to be required. Among these are the ability to monitor the execution of one's own actions, the ability to reason about actions that are outside one's own sphere of control, the ability to deal with actions which might (negatively) "interfere" with one another or with one's own goals, and the ability to form contingency plans to overcome such interference. Georgeff [Geo90] argues further that we will require an agent to be capable of coordinating plans of action and of reasoning about the mental state — the beliefs, goals, and intentions — of other entities in the world; where knowledge of other entities' *motivations* is limited or where communication among entities is in some way restricted, an agent will often have to be able to infer such mental state from its observations of entity behaviour. Kirsh, in addition, argues that for survival in real-world, human style environments, agents will require the ability to "frame and test hypotheses about the future and about other agents' behavior" [Kir91, page 177].

Now, while much of the behavioural repertoire needed for operating in complex dynamic environments is made available through a number of different reactive (layer $\mathcal{R}$) and planning (layer $\mathcal{P}$) control functions, situations will nevertheless arise in which a TouringMachine's initial goals — or the plans generated to achieve such goals — can no longer be carried out with success. As Minsky [Min86, page 163] puts it, "no matter how neutral or rational a goal may seem, it will eventually conflict with other goals if it persists for long enough." These exceptional situations, as noted in Chapter 3, are called goal conflict situations and they arise, in the TouringWorld domain, as a result of unexpected interference between two or more entities — for example, when the space-time trajectories of two agents intersect at a common point.[1] In such situations, the only viable or effective course of action will be for the agent to alter or dispose of some of its existing goals or plans and then to generate new ones which address the agent's new requirements and/or opportunities. Often, though, such exceptional situations will demand from agents a fairly sophisticated level of autonomous control: control providing a high degree of goal flexibility and which adds an enriched spatio-temporal dimension to the agent's decision making capabilities. In other words, control which would be very difficult to achieve solely with the kind of hardwired and pre-planned responses produced by a TouringMachine's reactive and planning layers.

6.2 Overview of Layer $\mathcal{M}$

The main objective of layer $\mathcal{M}$ is to provide an agent with the reflective and predictive — or *metaplanning* (see page 64) — capabilities which are necessary, or at least strongly desirable, for effective operation in complex multi-agent environments. More precisely, it provides the agent with a range of tools and functions for building and maintaining *mental* or *causal models* of world entities (including itself) which the agent can then use, for example, for identifying entity behaviours or any other world events which had not been expected, for causally explaining such observed behaviours and events, for detecting and re-

[1] Certain types of goal conflict will occur sooner or later in any reasonably interesting multi-agent TouringWorld scenario. As described in the previous chapter, a TouringMachine, ignorant at first of any other agents or of any objects other than the navigable paths that will lead it toward its destination, does not initially set out to stop at traffic lights or to give way to other agents at uncontrolled junctions; conflicts will occur, then, if the environment is populated with other such entities and if the agent is to obey certain traffic regulations and to avoid colliding with these other entities. More on this below.

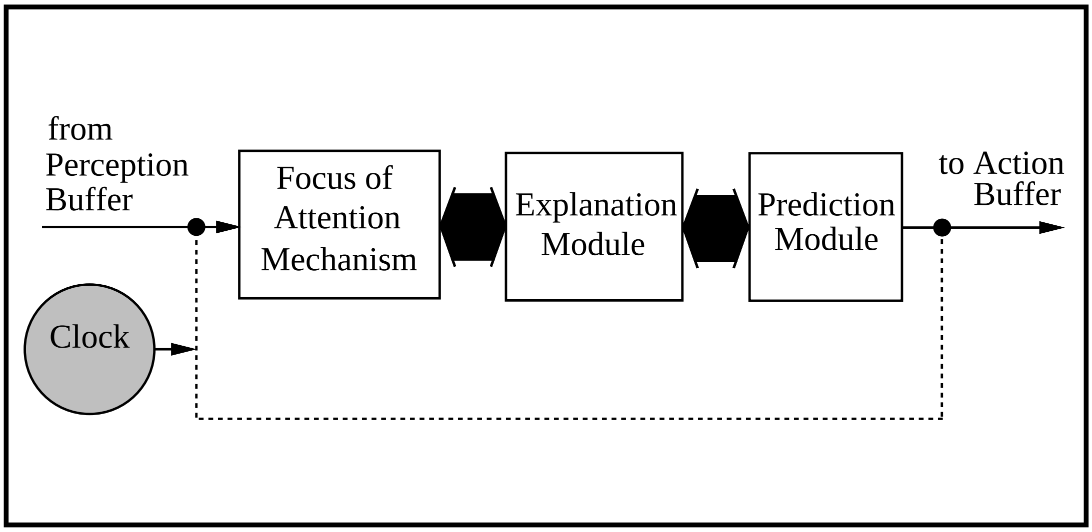

Figure 6.1: Top-level view of the modelling control layer $\mathcal{M}$.

solving any goal conflicts which might have arisen from these events (conflicts can occur within the agent's own set of goals or between the goals of different world entities), and also for making short- or long-term spatio-temporal predictions about entities' possible future behaviours and about the possible future outcomes of observed world events.

Layer $\mathcal{M}$, like layers $\mathcal{R}$ and $\mathcal{P}$, operates as an independent control module. In a synchronous fashion, $\mathcal{M}$ receives input from the agent's Perception Buffer at the start of a timeslice, and sends output to the agent's Action Buffer at the end of the timeslice (see Figure 6.1). In the TouringWorld domain, $\mathcal{M}$'s input takes the form of symbolic multi-attribute information records — see Figure 3.3 *(b)* — which describe the physical configurations of any perceived world entities, including that of the agent itself. Output takes the form of physical and communicative action commands for the agent's effectors and will be discussed in more detail below.

The functionality of this layer, as shown in Figure 6.1, is shared between three major components: a *Focus of Attention Mechanism*, the *Explanation Module*, and the *Prediction Module*. Like layer $\mathcal{P}$'s operations which could include potentially lengthy searches of the agent's plan space, the deliberative operations of layer $\mathcal{M}$ are also computationally expensive, certainly relative to those of layer $\mathcal{R}$. In order for TouringMachines to be capable of operating in dynamic real-time environments, then, the inter-operation latency of its various deliberative modelling functions must be bounded. One step toward achieving this is to limit the amount of information which layer $\mathcal{M}$ will have to reason about. This, as described in Section 6.3, is the main purpose of the Focus of Attention Mechanism.

The Explanation Module is responsible for generating plausible or inferred explanations about any entity behaviours which have recently been observed. More specifically, the Explanation Module is responsible for building and maintaining models of all observed entities (including itself) which are used in detecting *discrepancies* between these entities' current behaviours and those which had been anticipated or predicted to occur from previous encounters (see below). If any such behavioural discrepancies are detected, the task of the Explanation Module will then be to infer plausible explanations for their occurrence.

Once all model discrepancies have been identified and their causes inferred, the Prediction Module will then be responsible for constructing space-time projections or *simulations* for each of the entities being modelled. These are used by the agent to detect any potential interference or conflicts among the modelled entities' anticipated (or, in the case of its own model, desired) actions. Should any conflicts — *intra-* or *inter-agent* — be identified, the task of the Prediction Module will then be to determine how such conflicts might best be resolved, and also which entities will be responsible for carrying out these resolutions. Finally, in order for the agent to be able to make future comparisons between observed and anticipated entity behaviours, the Prediction Module will construct appropriate *expectations* for subsequent use by the Explanation Module, as mentioned above.

The potential gain in having the kind of explanatory and predictive powers which layer $\mathcal{M}$ provides, is that by having a better understanding of the causes behind the various entity behaviours which might be observed and by being able to make successful predictions about these entities' ensuing courses of action, a TouringMachine should be able to detect, and often resolve, potential goal conflicts early on — before they become irreversible. This, then, should enable a TouringMachine to make changes to its own plans in a more effective manner than if it were to wait for these conflicts to materialise. Before going any further into the operational details of layer $\mathcal{M}$'s explanation and prediction functions, however, a description of this layer's attention focussing capabilities will prove useful.

6.3 Focus of Attention

The purpose and general operational characteristics of layer $\mathcal{M}$'s Focus of Attention Mechanism are more or less identical to those of its layer $\mathcal{P}$ counterpart (see Section 5.3). The aim of the Focus of Attention Mechanism is to filter layer $\mathcal{M}$'s perceptual intake by reducing the set of all *perceived* world events to the set containing only those events which the agent (or the agent's

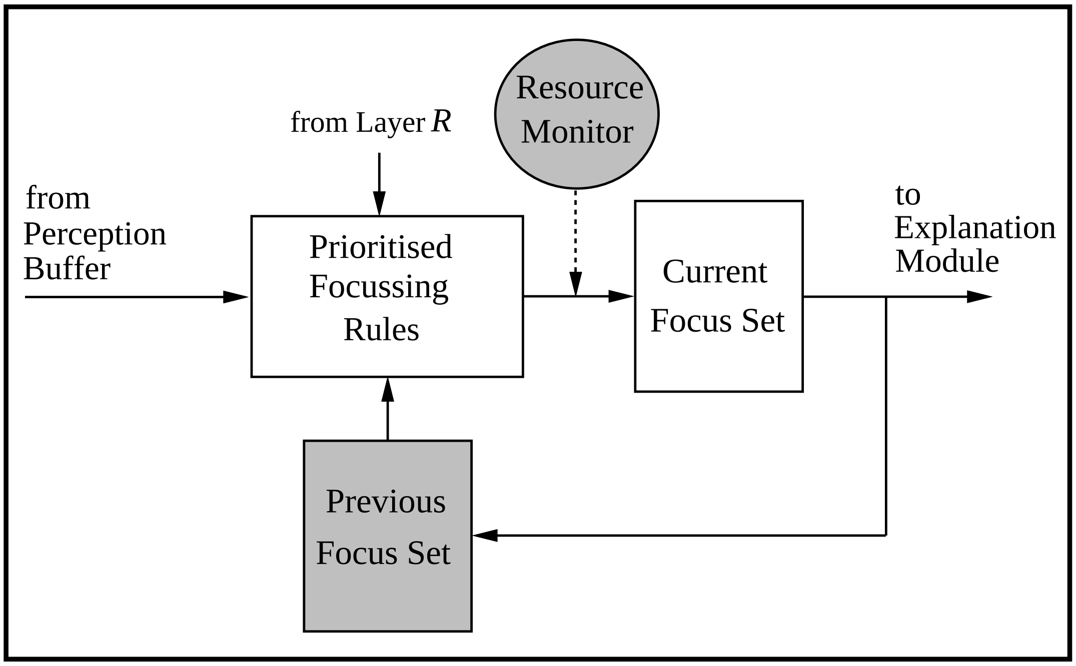

Figure 6.2: Layer $\mathcal{M}$'s Focus of Attention Mechanism

designer) considers *relevant* to its task-related needs. This is carried out to ensure that layer $\mathcal{M}$, which, like layer $\mathcal{P}$ is computationally resource-bounded, is capable of timely responses under a variety of real-time, changing world conditions.

Like layer $\mathcal{P}$'s attentional mechanism, layer $\mathcal{M}$'s (see Figure 6.2) consists of a set of user-specified, if-then *Prioritised Focussing Rules* which heuristically and selectively filter information from the agent's Perception Buffer, and which produce output in the form of a *Current Focus Set*: a collection of "relevant" perceptual information records — see Figure 3.3 *(b)*, for example — for subsequent use by the layer's Explanation Module. Applied from the point of view of the agent doing the focussing and always relative to the current world time, the focussing rules are used to express domain-specific relations about or between different entities which must be satisfied if these entities are to form part of the agent's Current Focus Set. The complete set of focussing rule predicates used in the TouringWorld domain is given in Figure 5.3. As entities alter their behaviours or move about in space or time, the relationships between the focussing agent and other world entities will change. When relationships change, different focussing rules will be satisfied and so different entities will be focussed upon as time progresses. The precise number of entities focussed on at any one time is established, as in layer $\mathcal{P}$, by "charging" from the Mechanism's per-timeslice, bounded, computational resource pool for each entity that is brought into the Current Focus Set.

Although the focussing rules used by layer $\mathcal{M}$ are selected by the agent's designer in advance, one mechanism exists for automatically altering the contents of this rule set at run-time. By allowing the agent's layer $\mathcal{R}$ to inform its layer $\mathcal{M}$ about the various entities it has been reacting to, this mechanism (which was alluded to in Chapter 4) enables layer $\mathcal{M}$ to take into account — that is, to include in its model set — types of entities which it might hitherto not have chosen to focus on. The implicit assumption being made here is that if the agent has just reacted to some entity it might well be that the agent's modelling layer does not possess a model of this entity (and that perhaps it should if it is to deal with this entity in a more reasoned and competent manner).

The presence of this rule-changing mechanism provides the main reason for maintaining two separate attentional mechanisms: one for layer $\mathcal{P}$ and one for layer $\mathcal{M}$. Since the individual focussing requirements of the agent's Planner and its Explanation Module are very likely to differ — each of the control layers $\mathcal{P}$ and $\mathcal{M}$ maintains a different model of the agent's world — it is important to ensure that the Current Focus Set which is used by one layer not be "biased" by the run-time focussing alterations that are made by the other. By providing each layer with its own Focus of Attention Mechanism this requirement is easily satisfied.

6.4 Explanation

6.4.1 Introduction

The Explanation Module comprises four main components: a *Message Handler*, a *Model Building Mechanism*, a *Model Discrepancy Handler*, and a *Theory Formation and Selection Mechanism* (see Figure 6.3). The Message Handler receives and processes messages which convey information about the status of other agent components. In particular, messages are received from the agent's effectors and from the agent's planning control layer $\mathcal{P}$. From the former it receives information describing which layer had been responsible for the agent's most recent action (this, as explained below, is useful for informing layer $\mathcal{M}$ that some unplanned — layer $\mathcal{R}$ — action has just taken place which, possibly, might work against layer $\mathcal{M}$'s longer term interests); from the agent's planning layer it receives an up-to-date statement of the agent's own current intentions — as described below, this is required as part of the agent's mental model of itself.

The Model Building Mechanism constructs and maintains mental or causal models of any entity whose corresponding information record presently resides

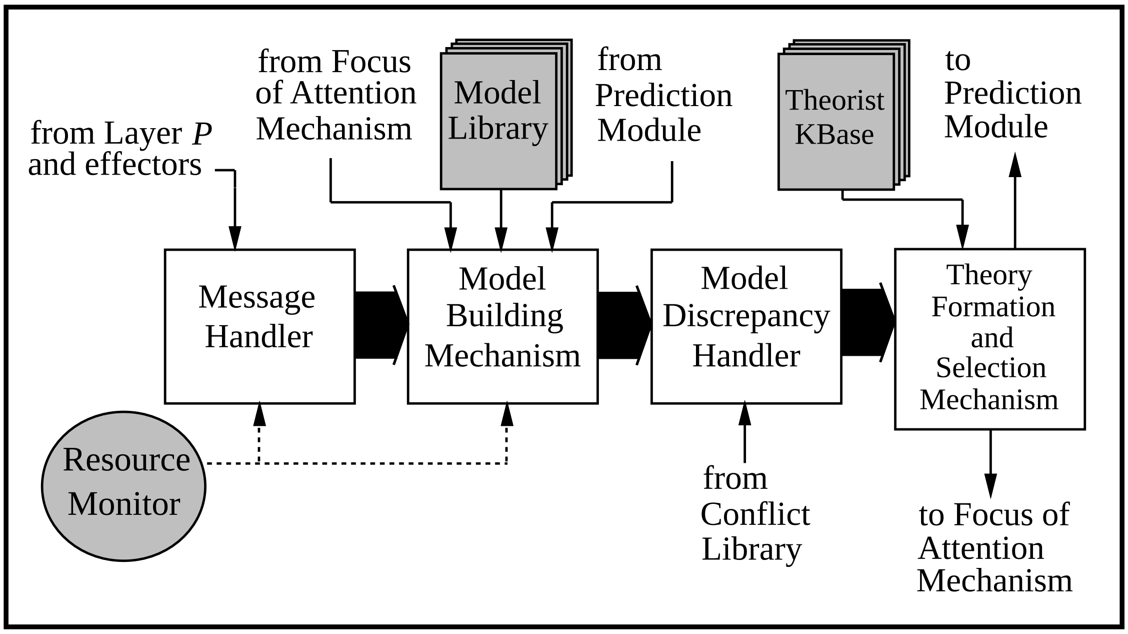

Figure 6.3: The Explanation Module

in the agent's modelling layer's Current Focus Set. In fact, in any given timeslice, only a subset of the entities which have been focussed upon are likely to be selected for modelling — the precise number being established through a process, not unlike that used by the Focus of Attention Mechanism, of "charging" from a per-timeslice resource pool for each entity under consideration.[2] Note that since the choice of which entities to model is being made on the basis of both the contents of and, more importantly, the specific ordering of entities within the agent's Current Focus Set, the Model Building Mechanism is effectively being driven by the same heuristic selection rules as the Focus of Attention Mechanism.

A model, as will be described more fully in Section 6.4.2, is a structure which is used by an agent for representing and reasoning about the physical and logical behaviour of any observed world entity. For the purpose of control and coordination, agents, as mentioned in Chapter 2, must be able to reason about their own and other entities' activities. In other words, agents must have the capacity to objectify particular aspects of their environment — that is, they must be able to construct and deploy internal models of themselves and other entities. Reasoning with models, then, is intended to allow agents to make meaningful interpretations of other entities' behaviours within the

[2]As described more fully in Chapter 7, the per-timeslice resource totals which are separately available for focussing on and modelling entities are defined by appropriate TouringWorld Testbed parameters.

situational context in which these behaviours are observed. The idea here is that by making correct interpretations, an agent will be provided with useful insights into other entities' future behaviours or likely changes in behaviour.

To an agent A, a model of entity E is a data structure which it can use for both posing and answering certain questions about E: for example, *"What are are E's current intentions?"*, *"Which one of E's goals is most threatened in the current conflict situation?"*, *"How will E resolve its most pressing goal conflict"*, or *"What will E be doing T units of time from now?"* A model, thus, can be regarded as a source of knowledge which contains references to and assumptions about those aspects of an entity which the modelling agent deems important for explaining the entity's behaviour.

The common abstract model structure which is used by all TouringMachines is a time-indexed 4-tuple of the form

$$\langle C, B, D, I \rangle$$

where C is the entity's *Configuration*, namely, (x, y)-location, speed, acceleration, orientation, and signalled communications; B is the set of *Beliefs* ascribed to the entity; D is its ascribed list of prioritised goals or *Desires*; and I is its ascribed plan or *Intentions* structure. C, B, D, and I are referred to as the variables or *components* of the model; it is from the specific values taken on by such components, as well as the specific connections which causally link such components together, that any inferences about entity behaviour are made (see below). Models, it should also be noted, are permanent structures which an agent will store and maintain throughout the duration of its operation. Thus, unless an agent deliberately chooses to discard its model of some world entity — this would typically be done if the agent had failed to observe the given entity for some considerable period of time[3] — all models held by an agent will persist from one timeslice to the next, being used and becoming updated as and when required. More on this in the following sections.

In the process of constructing a model of another entity, two cases need to be considered: the first case occurs when the entity to be modelled has just been observed by the agent for the first time; the second case arises when there exists a record of the entity having already been observed some time in the recent past. When an entity has just been observed for the first time, an initial model is built for it by retrieving a suitable *model template* from the *Model Library* (see Figure 6.3). Model templates are partially-instantiated data structures which serve as the basis for constructing models of individual world entities of various types (for example, TouringMachines, traffic lights,

[3]The precise length of time is established via a Testbed parameter called **ModelDiscardAfterTime** (see Chapter 7).

obstacles). Discussed in more detail in the following section, model templates provide a number of *default* values for one or more of the four components C, B, D, and I, which together constitute the entity model. For example, the model template for a TouringMachine entity might contain, among other things, the default desire `avoid-collisions`. Now, once a model has been retrieved for this new entity, two operations will take place. The first involves time-stamping the model with the current value, T, of the agent's internal clock (it will become clearer later why this is necessary). The second operation will be to update the retrieved model with any up-to-date information which might be held about the given entity. Among other things, the model will be updated with relevant information on the entity's current physical configuration: namely, its (x, y)-location, speed, acceleration, orientation, and its set of communicated signals; this information is extracted directly from the appropriate information record in the agent's Current Focus Set. In the second case, when the entity under consideration has previously been observed and, more importantly, previously been *modelled* by the agent, no library template retrieval takes place. Instead, the agent's existing model of the entity (which will have a time-stamp of $T-1$ if the agent's internal clock is currently at time T) will simply be updated using, as in the former case, appropriate configurational information obtained from the agent's Current Focus Set.

Thus far, the descriptions of how an agent's per-timeslice collection of models is determined and of how each model's configurational information gets filled in have purposely omitted a number of operational details. The first of these concerns the fact that, at times, layer $\mathcal{M}$'s Current Focus Set could be empty.[4] When this situation arises, the Model Building Mechanism will decide which entities to model by considering those entity models which it has stored from previous timeslices and then selecting from these whichever entities would have been selected had the entity names been obtained from the Current Focus Set. This is done simply by applying the Focus of Attention Mechanisms' heuristic prioritising focussing rules on the set of stored modelled entities. The second operational detail concerns situations in which, after selecting appropriate entity names from the Current Focus Set (assuming it is non-empty to start with), the Model Building Mechanism is left with extra resources for modelling other entities. In these situations, the procedure

[4] It is possible (although admittedly not very useful or common) to configure a TouringMachine so that is does not sense the world each and every timeslice — the agent's precise rate of sensing is established via the Testbed parameter **SensingRate** (see Section 7.4.3 and Appendix A) — but which nevertheless models and reflects about its surroundings at all times (a TouringMachine's rate of modelling is similarly established with a Testbed parameter, as described in Chapter 7). Now, when an agent does not sense during a given timeslice, it also, understandably, will not focus its attention on anything; to reflect this occurrence, the agent's Current Focus Set will be flushed empty whenever sensing is not performed.

is to consider selecting (resources permitting) any additional entities which do not appear in the Current Focus Set but which have nevertheless been modelled by the agent in the past. This also ties in with the third and final operational detail on retrieving and updating models: namely, how to update the configuration component of some entity's model — its location, speed, orientation, etc. — when the required information is not available from the Current Focus Set. Here, the solution is straightforward: the modelling agent will have to assume that the entity's *current* configuration is that which had been anticipated or *projected* by the agent's Prediction Module the last time this particular entity was modelled (more on this below). This information may well be incorrect but, given that the agent has no other means of knowing what each of the other entities is presently up to, this should serve as a viable default — certainly until the agent next senses and updates its Current Focus Set.

Now, once an agent's entire collection of stored entity models has been updated with the relevant configurational — model component C — information, the agent will then be in a position to exploit these models for the remaining processing stages in the Explanation Module: reasoning about entities' observed behavioural discrepancies and forming theories which explain the causes behind such discrepancies. However, before proceeding to give more detail on these two processing stages, it is worth mentioning one or two points about the role of resource monitoring in the modelling layer.

Just as the Focus of Attention Mechanism's operational costs were monitored to guarantee an upper-bound on its computational latency, so too are those of the Explanation Module (see Figure 6.3). In particular, the costs of such operations as message handling and model building/updating are continuously tallied and deducted from layer $\mathcal{M}$'s per-timeslice resource allowance. When resources run out, the Model Building Mechanism ceases to select any further items from the Focus of Attention Mechanism's Current Focus Set, effectively limiting, for the present timeslice, the set of entities to be handled by the remaining processing stages of the modelling layer: namely, model discrepancy handling, theory formation and selection, goal conflict detection and resolution, and model expectation generation (the latter two pertaining to the Prediction Module, as discussed in section 6.5). By limiting thus the number of entities to be processed, and by further ensuring that this precise number of entities will always be able to be processed within the time constraints imposed by the agent's constant-sized processing cycle — this can be done by ensuring that the (worst-case) time demands of each of the remaining layer $\mathcal{M}$ processing stages be taken into account by the resource tallying and deduction functions just referred to — the time-bounded responsiveness of the Explanation Module, in particular, and therefore of layer $\mathcal{M}$ as a whole, can be guaranteed.

6.4.2 Reasoning with Models

Model-based Reasoning (MBR) is an AI methodology for reasoning about — and more often than not, *diagnosing* — complex physical artifacts such as hydraulic or mechanical controllers or electronic circuits. More generally, MBR, or *reasoning from first principles* or *commonsense reasoning* as it is also known, is an umbrella term for studies covering many different aspects of advanced knowledge representation and reasoning [Lee89]. In particular, it concerns such concepts as causality and intention in order to reason about the way artifacts, systems, or indeed agents, perform.

The basic paradigm for model-based reasoning for diagnosis can best be understood as the "interaction of observation and prediction" [DH88, page 298]: given an artifact and a model of the artifact from which one can make predictions about the artifact's desired or expected behaviour, an *observation* will indicate what the artifact is actually doing, whereas a *prediction* will indicate what it is supposed to do. The most interesting event in this context, surely, will be any difference or *discrepancy* which occurs between the two.

Given a model of some artifact, diagnostic reasoning can be seen as the process of assigning credit or blame to parts or components of the model based on any behavioural discrepancies which have been observed. In situations where the model is presumed to be correct and where any model-artifact discrepancies can be regarded as indicating component malfunctions within the artifact, the diagnostic reasoning task is typically referred to as *troubleshooting*. Where, on the other hand, the artifact must be presumed correct and, conversely therefore, any model-artifact discrepancies are to be regarded as indications of "malfunctions" or required changes in the *model* of the artifact, the reasoning task is then known as *theory formation* [dKW86]. It is this latter task, the formation of theories about artifacts' or entities' behaviours, which is of concern in the modelling layer of a TouringMachine and which will be elaborated on shortly. Before that, however, it would be useful to provide some more detail on the structure and use of TouringMachines' model templates.

The abstract 4-tuple definition of a model template given above was purposely simplified to give a preliminary, high-level description of their basic structure and purpose. In fact, the models used by TouringMachines have a number of other components which should now be described. Model templates, as alluded to above, are frame-like structures which are defined, much like a TouringMachine's plan schemata, by a set of (**component**, `value`) pairs, as shown in Figure 6.4. Model retrieval is performed by searching the Model Library (see Figure 6.3) for any template whose **type** component value matches the type of the entity which is currently under investigation by the agent.[5] Entity types found in the TouringWorld domain include mobile agents (`touring-machine`), environment agents (`traffic-light`, `rain`, `fog`), as well as various objects (for example, `obstacle`, `wall`, and `kerb`); other types will be described in Chapter 7. The process by which TouringMachines model other TouringWorld entities which are not of type `touring-machine` — that is, entities which are either or both non-mobile and non-intentional — can be considered a special, and admittedly, less interesting case to describe. Although some such cases will be discussed below in Section 6.6, the remaining discussion will focus on the process by which TouringMachines model other TouringMachines (which includes, of course, the case of a TouringMachine modelling itself).

When a suitable type match is found, a resource check is carried out to determine whether the template that has been identified can be "afforded" by the agent. The value associated with the template's **cost** component is thus used to inform the Model Building Mechanism of the minimum number of per-timeslice resources it must have *before* this particular template can be processed (the purpose of resource charging was explained in the previous section). If this particular constraint cannot be satisfied, the Model Building Mechanism will continue to search for another suitable (cheaper) template of the same **type**. If one cannot be found, the Mechanism will then simply cease to model any more entities during the current timeslice. If, on the other hand, the **cost** constraint can be satisfied, the given model template will be selected and a number of its variables unified with appropriate data values supplied by the agent: in particular, the **name** component with the name of the entity which is being described with this model, and the **timestamp** component with the current value of the agent's internal clock.

The four most interesting components of any model have already been introduced above. These are the **configuration**, **beliefs**, **desires**, and **intentions** components. The **configuration** component of a model is used for storing information about the modelled entity's recently observed physical configuration: its last known (x, y)-location, speed, acceleration, orientation, and set of communicated signals (the latter is used to record, for example, whether the entity was last observed braking, honking its horn, or perhaps indicating to turn). Up-to-date configuration information, as mentioned above, is obtained from the modelling agent's Current Focus Set.

The **beliefs** component is used for storing what the agent considers to be the entity's own set of beliefs — in other words, the agent's beliefs about

[5]As with plan schema attributes, model template components and values are matched at retrieval time using standard Prolog unification [CM81].

```
type:  touring-machine
cost:  5
name:  EntityName
timestamp:  WorldTime
configuration:  [location, (X, Y)],
                [speed, Speed],
                [acceleration, Acceleration],
                [orientation, Orientation],
                [communications, Communications]
beliefs:  [size, size(self)]
          [physical-capabilities, physical-capabilities(self)],
          [sensing-parameters, sensing-parameters(self)],
          [focussing-parameters, focussing-parameters(self)],
          [layer-R-parameters, layer-R-parameters(self)],
          [layer-P-parameters, layer-P-parameters(self)],
          [layer-M-parameters, layer-M-parameters(self)],
          [current-focus-set, current-focus-set(self)],
          [rain-factor, 1.0],
          [fog-factor, 1.0],
          [inter-agent-distance, 3],
          [running-light-boundary, 2],
          [rights-of-way, rights-of-way(standard-uk)]
desires:  [avoid-collisions, []],
          [obey-regulations, []],
          [reach-destination, [[destination, Destination],
                               [travel-time, lt, Deadline],
                               [within-distance, Proximity]]
intentions:  [plan-a-route, Arguments]
defeasible components:  intentions
expectations:  [location, _],
               [speed, _],
               [acceleration, _],
               [orientation, _],
               [communications, _]
```

Figure 6.4: The `touring-machine` model template.

the entity's beliefs.[6] Each belief in the **beliefs** component is realised as a simple grounded (attribute, value) pair. In the template of Figure 6.4, for example, the pair `[inter-agent-distance, 3]` would represent the modelled entity's presumed belief that the recommended safety distance between agents (in motion) is 3 spatial units. Similarly, in the same template, the pair `[physical-capabilities, physical-capabilities(self)]` would represent
the agent's belief that the entity's physical capabilities (for example, its maximum speed, acceleration rate, etc.) are the same as those which the agent *itself* possesses (the argument `self`, in other words, refers to the agent doing the modelling and not to the entity which is being modelled).

The **desires** component of a model consists of a prioritised list of desires or goals which the modelled entity is considered to possess. As mentioned in the previous chapter, a TouringMachine may have a number of maintenance or *homeostatic* goals (for example, `avoid-collisions`, `obey-regulations`) and also a single *achievement* goal; in the TouringWorld domain this will always be `reach-destination`. The **intentions** component of a model is used for storing the entity's presumed intention structure: in other words, the time-indexed hierarchical plan structure which the agent believes the entity will, in the past, once have generated and, in the present, be processing in order to accomplish its sole `reach-destination` achievement goal. In fact, to simplify matters somewhat — and since all entities will be considered to have the same initial intention, `plan-a-route`, as well as identical Schema Libraries for use in decomposing this initial intention — a modelled entity's intention structure will be represented solely in terms of the entity's presumed *current* intention; that is, in terms of the hierarchical plan structure node which the entity's Planner is currently presumed to be processing. More on this below.

While it has been mentioned already that a common model template is used by TouringMachines for modelling other TouringMachines — and also, therefore, for modelling themselves — it is worth mentioning that the only concrete difference between self and non-self models will reside in the level of uncertainty of the information which they store. This arises from the fact that when a TouringMachine models another entity, it will never know with certainty what the entity's precise desires and intentions might be. Thus, even though entities are assumed to have a common set of prioritised desires and also the same initial `plan-a-route` intention, the precise arguments taken on by these model components — defining, among other things, each entity's particular target destination and task deadline — will remain unknown to

[6]When the entity being modelling is the agent itself, these beliefs can be regarded simply as the agent's beliefs (rather than its beliefs about its beliefs).

observers, typically throughout the duration of the task scenario. One distinguishing feature, then, between a TouringMachine's model of another entity and the typically complete and detailed model it holds of itself, will be the presence, in its non-self models, of many unbound component member values — see for example, `Destination` and `Deadline` in the (non-self) model template of Figure 6.4.

The process, then, of associating an initial, partially-instantiated model template to some entity, is essentially one of *ascribing* to the entity a default causal description or explanation of its behaviour. The very nature of TouringMachines as computationally and informationally resource-bounded agents would suggest, therefore, that much of what they might ascribe to other entities is unlikely to be completely correct. Indeed, apart from the information they possess about (and ascribe to their own models of) themselves — in other words, about their own physical configurations, their own beliefs, desires, and intentions — plus the physical configuration information which they can acquire about other entities from their sensory and focussing mechanisms, the rest — that which TouringMachines cannot observe in other entities directly — may well be incorrect. For models to be useful, then, it would appear that any agent which makes use of them must, at times, be willing to revise them whenever they disagree with the agent's current observations of the world.

When undertaking the revision of a model, any *number* of the model's components might be considered "culprits" behind the observed model-entity discrepancies and therefore as being potentially worth revising. Also, different *orders* of such component revisions might also be worth considering. This is precisely the sort of information which is stored in the **defeasible components** attribute of a model (see Figure 6.4): namely, the ordered list of model components which, upon the detection of any model-entity discrepancies, the agent should consider revising. Now, in general, when attempting to explain some complex entity's behaviour, the question of what assumptions might need to be revised and what assumptions should not, is by no means straightforward to answer. While some aspects of this question are addressed again in Chapter 9, it is beyond the scope of this dissertation to provide a general theory for modelling complex intentional behaviour in autonomous agents. To simplify matters in this respect, therefore, this dissertation is concerned solely with modelling agents which can be best be described as being mentally and structurally *homogeneous*; in the TouringWorld domain, this means that agents — TouringMachines — will be assumed to have universally similar physical configurations, beliefs, and desires. In other words, TouringMachines will be assumed to differ only with respect to their intentions. How one might deal with *heterogeneous* agents — those which may or may not share the same goals and beliefs as every other entity in the environment — is raised as an avenue for future work in Chapter 9. The next two subsections will develop further the notions of model-entity discrepancy and intention ascription as theory formation.

6.4.3 Handling Model Discrepancies

The philosophy behind ascribing default models to other world entities is similar to that which lies behind the use of the *default ascriptional rule* in the ViewGen system of Wilks and Ballim [WB87, page 119]: namely assume that one's view of another entity's view is the same as one's own "except [when] there is explicit evidence to the contrary." To an agent which is modelling its environment, the evidence that some entity is not behaving as expected will manifest itself in the form of a model-entity difference or discrepancy. This, of course, raises the questions of where, when, and how such behavioural expectations get formed. As discussed briefly at the start of this chapter, expectations about other entities' behaviours are generated by the agent's Prediction Module. The precise mechanism behind the generation of these expectations will be given below in Section 6.5. For now, it is sufficient to note three things about behavioural expectations: *(i)* any expectations held by an agent at time T will have been made using stored model information from time $T-1$: in other words, information obtained from appropriate models displaying a **timestamp** component of $T-1$; *(ii)* although framed within the context of the entity's presumed beliefs, desires, and intentions, expectations about any given entity will refer solely to that entity's projected physical configuration: its expected (x, y)-location, speed, acceleration, orientation, and communicated signals; and *(iii)* such projected physical configurations will be stored as models' **expectations** components (see Figure 6.4).

Discrepancies in a given entity's behaviour are detected through direct comparison of the values in the (current) **configuration** with the values in the (projected) **expectations** components of that entity's model. In particular, a discrepancy will exist if any of the entity's current location, speed, acceleration, or orientation parameter values differ from those previously projected and/or if the entity's current set of communicated signals is not identical to its previous set. Thus, much like envelopes in the Phoenix agent [HHC90] or monitors in the Dynamic Reaction architecture [SH88] (see Section 2.4), expectations act as constraints on the outcomes of actions: outcomes which, in the case of an agent modelling itself, were *desired* by the agent to occur; or which, in the case of the agent modelling some other entity, were *predicted* to occur. In this respect, then, a model discrepancy can be regarded as an indication that the model of the entity under consideration is in some sense "faulty" and so should somehow be revised. Note, though, that when an agent detects discrepancies in the model it holds of itself, there is not so much a need for it to revise this self model — after all, the agent knows its own beliefs, desires, and intentions with absolute certainty — as there is for checking whether the discrepancy, which is likely the result of one of the agent's actions not executing as intended, might be an indication that one or more of the agent's own goals are now under threat. More on this below.

It should be pointed out at this stage that detecting a discrepancy between actual and predicted (or desired) behaviours need not on every occasion force the agent into a wholesale revision of its faulty model. This is because associated with each of the parameters of a model's **expectations** component are upper- and lower-bounds on the deviations which are permitted before a model revision becomes necessary. These deviation or *tolerance bounds* — **ModelLocationBounds**, **ModelSpeedBounds**, **ModelAccelerationBounds**, and **ModelOrientationBounds** — are implemented as Testbed parameters and so can be set to any size by the user (more on this in Chapter 7). Different settings for these tolerance bounds are likely to affect both the amount of environmental change that the agent will perceive and the amount of time the agent will need to spend revising its models. This issue of "environmental sensitivity" is an important one and will be investigated more closely in Chapter 8.

After checking each one of its models for any discrepancies, an agent will be in one of two situations: either it will have found one or more discrepancies in one or more of its models, or it will have found none. To cope with the fact that TouringMachines are resource-bounded and that, in a dynamic multi-agent world, it is probably more important that they respond to potential conflicts with other entities than make minor adjustments to their own travel speeds so that they can satisfy their pre-imposed deadlines, the TouringMachine agent architecture is designed to favour dealing with discrepancies found in the models of other entities over those which might have arisen in the agent's self model. In other words, TouringMachines are designed to favour re-explaining other entities' behaviours over checking whether their own goals are under threat. These two types of situations will now be considered in more detail, starting with the situation where the agent finds no discrepancies.

In situations where no model discrepancies have been found, the agent, in effect, takes the opportunity to check whether any of its own prioritised goals or **desires** are under threat — whether in the present or in the longer term. Now, since the agent did not detect any self model discrepancies — in other words, since the agent appears to be moving along as it predicted it would — it might seem redundant for the agent to check whether any of its goals are in danger of not being met. There are, however, situations

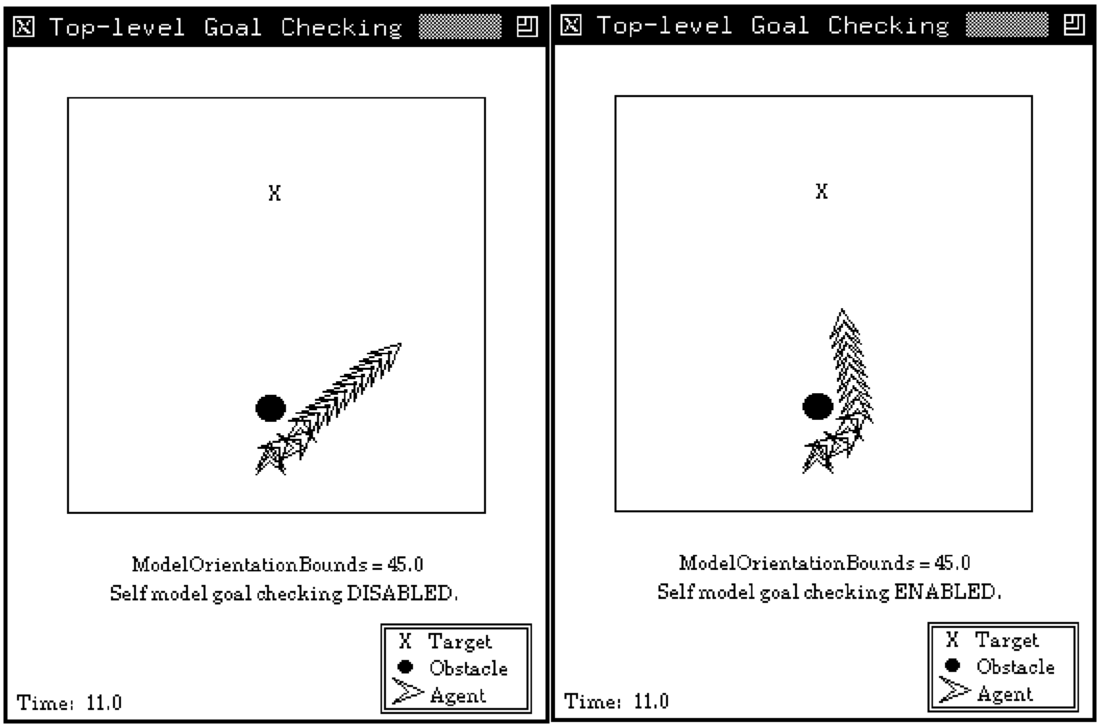

Figure 6.5: With reasonably wide tolerance bounds on its expected orientation — for example, +/-45° — and with self model goal checking disabled, it is possible, so long as no individual deviation causes the agent to move off its original heading by +/-45°, that the agent will fail to notice any orientation discrepancies and eventually end up seriously off course (see left-hand frame). If, on the other hand, the agent is able to carry out self goal checking (one assumes also that the agent's `reach-destination` goal has a suitably constraining `within-distance` argument attached to it — see Figure 6.4), then the agent will be less likely to stray off course to the same extent (see right-hand frame).

where such goal checks could prove crucial to the agent's long-term success. Consider, for example, an agent which has reasonably wide tolerance bounds placed on its expected orientation, say **ModelOrientationBounds** = +/-45°. Parametrised thus, the agent, upon reacting for example to the presence of some obstacle in its path, could easily end up heading away from its goal — terminally, without ever noticing any long-term problem — so long as no individual reaction causes an orientation swing greater than 45° either side of its original heading (see Figure 6.5). Goal conflicts of this type can only be detected by considering, in turn, each of the agent's prioritised goals — that is, each set member in the **desires** component of the agent's self model (see Figure 6.4) — and running it through appropriate *conflict resolution rules* residing in the agent's *Conflict Library* (see Figure 6.3). The purpose of these rules is to enable the agent to project any of its goals in space-time and to determine whether there exist any potential conflicts — either in the short- or long-term — vis-à-vis their eventual achievement. A detailed description of these rules and an explanation of how they are used will be deferred until Section 6.5.4 where the agent's Goal Conflict Detector will be discussed. For present purposes, it is sufficient to point out that if, upon application of its various conflict resolution rules, any of the agent's goals are considered to be under threat, a record will be made of this fact so that the agent's Goal Conflict Detector can subsequently take some appropriate action. More on this later.

Now, in situations where the agent *has* detected one or more model discrepancies, three cases must be further distinguished. The first involves a very special type of discrepancy which occurs exactly once — namely, when the agent first decides to model itself. This discrepancy will arise by virtue of the fact that the **expectations** component of the agent's self model is empty (this model has yet to be processed by the Prediction Module so no expectations will have been generated thus far). This discrepancy, in fact, will be interpreted as a flag instructing the Explanation Module to send a message to the agent's layer $\mathcal{P}$ setting out the initial task to be taken on by the Planner; the task, `plan-a-route` plus associated arguments, is obtained from the **intentions** component of the agent's self model (this messaging procedure was described earlier in Section 5.4.2). The second discrepancy detection case, like the first one, also involves the detection of a self model discrepancy, although this time it arises not because the agent's model is new but because there is some deviation between its own **configuration** and **expectations** components. As far as attempting to explain such self model discrepancies, little is done at this stage except to make a record of the discrepancies which have occurred and to defer treatment of them until the Goal Conflict Detector is invoked (see below in Section 6.5.4). The final case involves the detection of a discrepancy in the model of some other entity. This will require that the agent revise its model of the particular "errant" entity in an attempt to explain this entity's new and unexpected behaviour. This model revision process will be the subject of the next subsection.

6.4.4 Theory Formation and Selection

An agent, upon deciding to revise its model of some world entity, will first have to consider which of the model's components need to be changed. As mentioned in Section 6.4.2, this problem has been simplified somewhat in this dissertation by restricting the **defeasible components** of an entity's model to consist solely of the entity's presumed intentions — in other words, the plan structure which the modelling agent believes the entity to be processing in order to accomplish its single achievement goal `reach-destination`. Now, in this context, the process of revising a model can usefully be viewed as one of diagnostic reasoning or theory formation: namely, the identification of model "malfunctions" from observations of the modelled entity. It turns out that a suitable and practical system with which to carry out such diagnostic reasoning already exists and is called Theorist.

Theorist [PGA86] is a logic programming system for constructing *scientific theories* — that is, for constructing explanations of *observations* in terms of various *facts* and *hypotheses*. Theorist is a system for both representation and reasoning. A Theorist knowledge base consists of a collection of first order clausal form logic formulae which can be classified as: *(i)* a closed set of consistent formulae or facts, F, which are known to be true in the world; *(ii)* the possible hypotheses, Δ, which can be accepted as part of an explanation; and *(iii)* the set of observations, G, which have to be explained. Given these, the Theorist reasoning strategy attempts to accumulate consistent sets of facts and instances of hypotheses as explanations for which the observations are logical consequences. An explanation or *theory* is then a subset of the possible hypotheses which are consistent and which imply the observations. More formally, G is said to be *explainable* if there is some subset D of Δ such that

$$F \cup D \models G \text{ and}$$
$$F \cup D \text{ is consistent.}$$

D is said to be a *theory that explains* G. D should then be seen as a "scientific theory" [PGA86, page 4].

Theorist has been described as both a theory and an implementation for default and *abductive* reasoning [Poo88]. One of the several ways in which Theorist can been used, then, is for performing *abductive diagnosis*; namely, finding a set of causes (for example, diseases) which can imply the observed effects (for example, patients' symptoms). Now, by taking the system or artifact that is being diagnosed as the entity that our agent is modelling, and by re-interpreting "symptoms" as the entity's observed actions, then the causes behind this entity's actions — be they physical or communicative — can be regarded as the entity's intentions. (To emphasise, once more, TouringMachines ' beliefs and desires are accepted as being common and so will not be considered in the theory formation process.) Note, then, that, in the context of TouringMachines, the process of finding the intentions which are the cause of some other entity's actions is effectively one of performing *plan inference* or *recognition* [Car90b].[7] A brief description of how Theorist has been applied to intention ascription or plan recognition in the TouringWorld domain now follows.

[7] Davis also refers to the task of inferring an agent's goals and plans from its actions as "motivation analysis" [Dav90, page 395].

Theorist is invoked once for every one of the agent's entity models that displays a model-entity (or expectation-observation) discrepancy. In particular, Theorist is called by supplying it with the name of the agent that is doing the modelling, the name of entity that is being modelled, the agent's observations of that entity (that is, all relevant details of the entity's current configuration as modelled by the agent), and the current value of the agent's internal clock. Theorist's reasoning strategy then tries to accumulate consistent sets of facts and instances of hypotheses, or defaults, as explanations for which the observations are logical consequences. The facts and defaults reside in the *Theorist KBase* (see Figure 6.3), a knowledge base containing a domain model of the TouringWorld expressed in terms of the various "faults" that can be used to explain entities' "errant" behaviours. In the present context, faults can be viewed as the causes for — or the intentions behind — why certain events — or certain observed actions of some entity — might have occurred in the world.

Causal knowledge of the TouringWorld domain, then, is represented in the Theorist KBase as implications of the form *intention* $\Longrightarrow$ *observations*. Figure 6.6 gives a selection of those used by TouringMachines in the TouringWorld domain. The syntax used here for representing facts and defaults differs only slightly from that given by Poole *et al.* [PGA86] and is the following:

`fact:` $\langle clause \rangle$;
`default:` $\langle name \rangle \langle clause \rangle$;

where $\langle clause \rangle$ is a first order clausal form logic formula. The first of these statements means that the clause is a member of the set of facts F (defined above); the second means that for every instance of the name, the clause is a member of the set of possible hypotheses or defaults Δ. The name, which exists primarily as a way of referring to the default, can be used in a theory to explain the observations G.[8]

After the observations of a given entity have been processed by Theorist, the modelling agent will find itself in one of three possible states vis-à-vis being able to explain the modelled entity's intentions. The first comes about when the agent fails to produce any explanation at all for the entity's behaviour — in other words, when Theorist fails to generate any theory which can account for the entity's observed actions. This might occur, for instance, if the domain model described by the Theorist KBase were incomplete. The approach adopted in this case is to have the agent ascribe the "intention" `can't-explain` to the observed entity. An entity which has been ascribed this

[8]Names like `follow-path` and `start-overtake` in Figure 6.6 are exactly the names of the schemas which reside in each TouringMachine's Schema Library (as described in the previous chapter). The full list of these names is given in Appendix A, page 194.

```
fact:  intends-to-follow-path ⟹ C = ∅, S > 0;
fact:  intends-to-start-overtake ⟹ 'signal-right' ∈ C, S > 0;
fact:  intends-to-finish-overtake ⟹ 'signal-left' ∈ C, S > 0;
fact:  intends-to-stop-at-light ⟹ 'braking' ∈ C, A ≤ 0;
fact:  intends-to-stop-at-junction ⟹ 'braking' ∈ C, A ≤ 0;
fact:  intends-to-turn-to-target ⟹ 'signal-right' ∈ C, S = 0;
fact:  intends-to-turn-to-target ⟹ 'signal-left' ∈ C, S = 0;

default:  follow-path intends-to-follow-path;
default:  start-overtake intends-to-start-overtake;
default:  finish-overtake intends-to-finish-overtake;
default:  stop-at-light intends-to-stop-at-light;
default:  stop-at-junction intends-to-stop-at-junction;
default:  turn-to-target intends-to-turn-to-target;
```

S, A, and C are the values associated, respectively, with the speed, acceleration, and communications members of the **configuration** component of the modelled entity.

Figure 6.6: A selection of Theorist KBase entries used in the TouringWorld domain.

intention will be treated by the agent with special care; specifically, in subsequent timeslices, the agent will treat the presence of a can't-explain value in the **intentions** slot of its model of some entity as a "reminder" that no theory had previously been found which explained this entity's behaviour and that, as a result, theory formation should be attempted again in the hope that the entity's behaviour might have since become recognisable. This procedure, in fact, will be repeated until the entity's behaviour can be properly explained. The second state comes about when Theorist has found exactly one theory — that is, one intention name — which adequately explains the entity's behaviour. In this case, the intention name is copied into the **intentions** slot of the agent's model of the entity, this model then being ready for use by the agent's Prediction Module (see next section).

The third state an agent can find itself in occurs when it has found several (two or more) plausible and consistent theories which explain the given entity's observed behaviour. This might occur, for instance, if there existed insufficient sensory information (observations) to disambiguate the potential causes behind the modelled entity's actions. For example, if an agent, with the domain model of Figure 6.6, observed some stationary or decelerating entity with its brake lights currently on, it would be hard pressed, certainly without

```
while sufficient resources exist do process pending input messages
M_T := ∅
while sufficient resources exist and
        current focus set ≠ ∅
    do M_T := M_T ∪ next entity from current focus set
while sufficient resources exist
    do M_T := M_T ∪ next entity from M_{T-1}
discard any out of date models for entities remaining in M_{T-1}
foreach Entity ∈ M_T do begin
    if Entity ∉ M_{T-1} and
      model library has appropriate template
    then retrieve template and create model for Entity
    if sensing performed during current timeslice T
    then update Entity's modelled configuration with observations
    else update Entity's modelled configuration using expectations from T − 1
    identify any discrepancies between observations and expectations
    foreach discrepancy identified do explain discrepancy
end
if no discrepancies identified then check self goals
```

Figure 6.7: Algorithm to perform model-based explanations of entity behaviour (pseudocode). $\mathcal{M}_T$ is the agent's collection of stored entity models at time T.

further information, to know whether the entity's intention is `stop-at-light` or `stop-at-junction` or perhaps even something else. In the current implementation of the TouringMachine Explanation Module, little account is taken of this fact, a single and final theory being chosen randomly in cases where more than one have been generated. With such a strategy, an agent will often ascribe an entity the wrong intention, possibly leading the agent into new — and also increasingly threatening — conflict situations until the entity's behaviour can be unambiguously explained. Now, while in many of these situations a TouringMachine's reactive capabilities will, at the very least, probably save the agent from terminal damage, in general, an agent would be expected to benefit greatly from being able to ascribe intentions correctly the first time around. There are a number of specific enhancements one could make in this respect: for instance, an improved theory selection mechanism could be designed which could take account of entities' previous actions and intentions, which could differentially represent and reason about the disparate sources of evidence used in deriving explanatory theories (for example, observed actions, communicated intentions, default assumptions about rights-of-way and other traffic regulations), or which could be made to consider the expected utility of each entity's different possible outcome states and then calculate the probabilities with which each ascribable intention enables the entity to achieve such states. In fact, a number of possible solutions to this problem have been reported elsewhere in the AI literature and include, among others, uncertainty reasoning via symbolic endorsements [SC85], Dempster-Shafer evidential reasoning [Car90a], probability theory [GN87, pages 177–186], and decision theory [DW91, pages 265–279]. Further consideration of these techniques must at present, however, remain as a possible avenue for future work.[9]

To conclude this subsection, the full pseudocode algorithm implementing the various processes of the Explanation Module is given in Figure 6.7.

6.5 Prediction

6.5.1 Introduction

Prediction is the process of reasoning about the anticipated relations between a system and its environment in order to determine the course of action the system should follow. Thus, unlike *feedback* processes which monitor behavioural discrepancies in order to refine a system's actions (a TouringMachine's Explanation Module, for example, is a feedback process), the process of making predictions is one of *feedforward*: prediction involves monitoring the system's environment directly and applying appropriate compensatory signals to the system *before* waiting to receive feedback on how the system's performance has been affected by particular disturbances or changes in the environment. One advantage of feedforward Gregory [Gre87, page 260] argues, is speed: environmental changes can be compensated for before they have any noticeable effect on the controlled system's behaviour. The price paid for this, however, is in added controller complexity: the controller must have reasonably accurate models of the various effects such environmental changes and events can have on the system. The models needed by a TouringMachine for making predic-

[9] In fact, in addition to researching issues on theory selection, there are a number of issues concerning theory formation or plan recognition which have also been ignored in this dissertation but which are worthy of further study: for example, how to deal with an entity which has multiple intentions or which is executing sequences of (possibly simultaneous) actions which are interleaved with actions from its other plans, how to recognise flaws in the plans of observed entities, how to deal with entities whose plan spaces differ from those of others, or how to deal with an entity whose knowledge of others' plans is partial or whose beliefs of others' plans differ from those held by other entities. A number of these issues have been or are currently being addressed by other researchers in the field; Carberry [Car90b] provides an excellent review and analysis of much of this research.

tions about its world are precisely the kinds of entity models which have been described above.

Making predictions in a dynamic multi-agent world, Dawkins [Daw76, pages 59–64] argues, is a chancy endeavour and any decisions an agent — or "survival machine" — might make as a result of its predictions will always, to a degree, prove something of a "gamble". Short of possessing the capacity to learn from past behaviours and thereby to associate current situational contexts with expected outcomes, a TouringMachine, through its set of stored causal models of the different entities inhabiting its world, is able nevertheless to make predictions through a process of event *simulation*. A TouringMachine's Prediction Module is designed to realise such a process and will now be described.

6.5.2 Overview

An agent's Prediction Module comprises three main components: a *Message Handler*, a *Goal Conflict Detector*, and a *Model Expectation Generator* (see Figure 6.8). The Message Handler receives and processes messages which convey information about the potential conflict status of other agent components. In particular, messages are received from the agent's effectors and from the agent's planning control layer $\mathcal{P}$. From the former it receives warnings that the agent has attempted to execute some action which has caused some physical capability conflict to occur: for example, the agent might have attempted to execute a `change-speed` action which, if carried out in full, would cause the agent to exceed its maximum speed capability. From layer $\mathcal{P}$ it receives warnings that the Planner has failed in one of its operations: for example, in decomposing a composite schema, the Planner may have attempted to process a child schema for which no match can be found in layer $\mathcal{P}$'s Schema Library. Such warnings will be treated as indicators that one or more of the agent's goals could be threatened or be in a state of possible conflict: for example, failing to carry out a `change-speed` action in full or failing to retrieve a required schema may have a subsequent impact on the agent's ability to reach its target destination on time. These warning messages will thus be forwarded to the Goal Conflict Detector for further consideration.

A TouringMachine agent, ultimately, is motivated to act by the need to accomplish or satisfy each and every one of the goals which resides in the **desires** component of its self model. Whether the agent's goals are of the homeostatic type — those which are achieved continuously throughout the agent's operational existence — or of the achievement type — those which are explicitly planned for and (if all goes well) eventually terminated upon

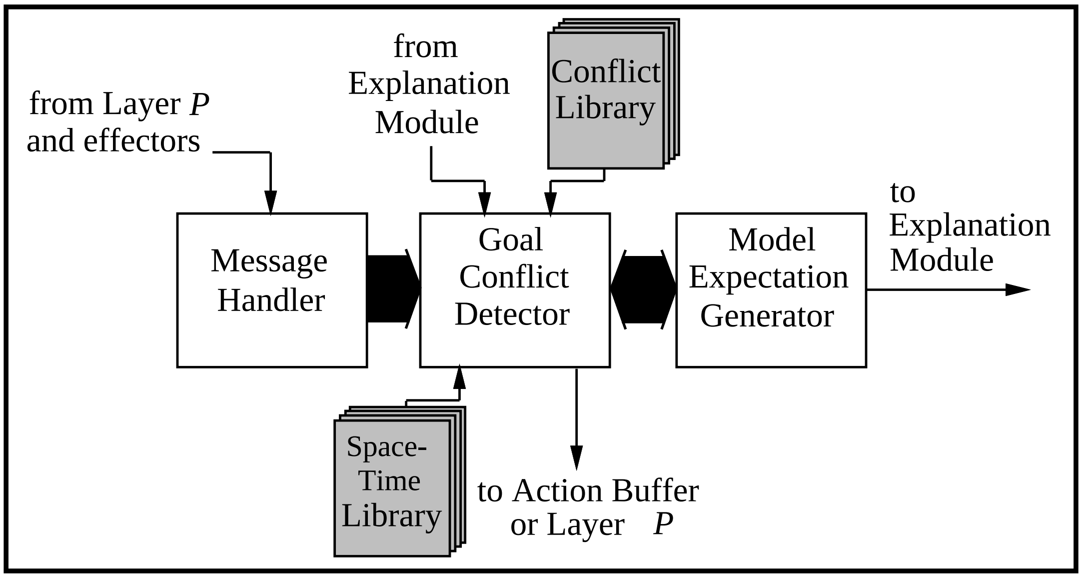

Figure 6.8: The Prediction Module

their successful completion — the Goal Conflict Detector's task will be to ensure their protection against any exceptional events which have either just recently taken place or which the agent, after a process of generating model-based projections of future world events, has predicted will take place. These exceptional events or conflicts which the Goal Conflict Detector must deal with can originate from a number of sources, both from within the agent itself and as a result of certain unexpected interactions between the other entities in the world. Resolving such intra- and inter-agent conflicts will often require some form of action or change of plans on the part of the agent and/or the other entities. More on this below.

Once all existing and predicted goal conflicts — within and without the agent — have been identified, the Model Expectation Generator, taking into account any actions or change of plans which might have been proposed as possible resolutions to the identified conflicts, will be responsible for generating a description of the agent's — and all other world entities' — future expected trajectories. As described in Section 6.4, descriptions of entities' predicted physical configurations will be used by the agent's Explanation Module to detect discrepancies between the models it possesses of the entities and the corresponding observations it subsequently makes of these. Model expectations are generated using the same basic model projection techniques which are employed by the Goal Conflict Detector when looking for potential goal conflicts. A description of these techniques should prove useful before discussing the handling of goal conflicts and generation of model expectations.

6.5.3 Projecting Models

A TouringMachine constructs spatio-temporal projections of another agent's behaviour by performing a "cognitive" or *knowledge level* simulation of the agent's mental state and consequent action sequences: in other words, by taking to itself "the role of the other (i.e. the agent), assuming the agent's goals and attending to the common external environment, [such that] the actions it determines for itself will be those that the agent should take." [New82, page 109]. The purpose of a TouringMachine making these projections, is to determine, in the light of the present state of the world, whether any agent — and, indeed, whether the TouringMachine itself — is destined for some future conflict with any of the other entities in the world. The conflicts in question here, are those which affect the successful completion and/or continued satisfaction of the TouringMachine agent's various goals: `avoid-collisions`, `obey-regulations`, and `reach-destination`.

A spatio-temporal projection of some entity is a projection of this entity's expected movements through space over a given period of time. TouringMachines, as we know, construct models of other entities which, among other things, can be used for storing descriptions of these entities' current physical configurations. With such models at its disposal, it would be relatively easy for a TouringMachine to construct projections of each of the entities in its model set: since entities are assumed — see Chapter 7 — to move with uniform acceleration, following trajectories appropriately constrained by the quadratic motion equations $s = v_0t + \frac{1}{2}at^2$ and $v = v_0 + at$, a simple way for the agent to project another entity's movements between the present time, T_0, and some future time T_1, say, would be to take the modelled entity's present (T_0) configuration values, feed these into the appropriate kinematic equations above (setting $t = T_1 - T_0$), and then computing the entity's new configuration — namely, its (x, y)-location, speed, acceleration, orientation, and communicated signals at time T_1. This is what might be called making the *inertial assumption*.

As argued above, making predictions of future activity based on present information alone is a potentially risky endeavour, especially in a dynamic domain like the TouringWorld. So much so, in fact, that even where it might be possible to make predictions with some degree of certainty, it is unlikely that these predictions could be relied upon over extended periods of time.[10]

[10]Besides the length of predictions, other issues such as the degree or speed of environmental change are also likely to affect the long-term reliability of any predictions an agent might

It would seem, then, that to ensure that predictions prove as useful as possible, the agent making them should use as much (relevant) information about the world — and, in particular, about its set of modelled world entities — as it possibly can. In the TouringWorld domain, it turns out, a number of information sources readily suggest themselves: the modelled entity's current configuration; the entity's ascribed beliefs, desires, and intentions; and, also, information on any relevant environmental constraints which might impact on the entity's possible behaviours (for example, the presence of information signs, path junctions, kerbs, walls, and, most importantly, other mobile intentional agents). How these various information sources come into play when projecting entities' actions and also when detecting and resolving potential goal conflicts will now be elaborated on.

In projecting the potential actions of some modelled entity, a TouringMachine, in addition to considering this entity's current physical configuration, also takes into account the entity's ascribed intention. Now, whereas projections based on the inertial assumption could be readily computed via a simple application of the appropriate kinematic equations, consideration of an entity's intention will require slightly more sophisticated spatio-temporal projection capabilities. To assist in this process, a TouringMachine, through its *Space-Time Library* (see Figure 6.8), is given access to a number of domain-dependent spatio-temporal projection functions. Functions are of the form:

$$project(Configuration, Intention, TimePeriod) : Projection$$

where $Configuration$, a vector of the form $\langle (x, y), v, \dot{v}, \theta, comms \rangle$, describes the projected entity's current location, speed, acceleration, orientation, and communicated signals set, respectively; $Intention$, the projected entity's ascribed intention; $TimePeriod$, the period of time over which the projection is to be made;[11] and $Projection$, the computed path which the entity will be expected to follow over the given time period.

The path followed by an entity which is moving inertially could, as explained above, be determined by simply computing, for the given period of projection, a single quadratic motion trajectory starting from the entity's initial location. Computing the path followed by an entity with a "complex" intention will be slightly more involved. In particular, to project an entity whose currently ascribed intention can be expected to cause this entity to take some action(s) within the time period over which the projection is being made, the

make. The issue of trading off reliability for efficiency in predictions will be considered more closely in Chapter 8.

[11] The length of time over which projections are made is defined via the Testbed parameter **ConflictDetectionHorizon** — see Chapter 7.

paths computed by a TouringMachine — in other words, the *Projection* results generated by the TouringMachine's spatio-temporal projection functions — will in fact be handled as *composite* trajectories: specifically, lists made up of any number of "distinctive" temporally bounded *sub-trajectories* which the modelled entity might be expected to follow. This is best illustrated with an example.

Consider an entity which has been ascribed, by some agent, the intention `stop-at-light`. What this means from the point of view of the agent is that, in order to protect one or more of its goals (in this case, `obey-regulations` — see below), the entity will have adopted (at some stage in the past) the intention to stop at the set of traffic lights which it is currently approaching. Assume the agent constructs entity model projections which cover a period of time P beyond the agent's current clock time of, say, T_0. Now, if the agent, in projecting the entity's movements, anticipates that the entity will come to stop near the identified traffic lights at some time T_1 which is contained *within* the time interval P, the agent, if it is to build an accurate prediction of the entity's behaviour beyond time T_1, will need to view the entity as following a new and separate motion (sub-)trajectory from the one that it was following before reaching the traffic lights; thus for example, the agent might, after time T_1, predict that the entity will remain stationary until the end of the projection period — that is, up to time $T_0 + P$.

A path computed by an agent for some entity, then, is the set of sub-trajectories which the agent expects the modelled entity to follow, with each sub-trajectory having a specific duration (and implicit start and end times) and corresponding to some change — typically resulting from the entity having taken some action — in the physical configuration of the entity being modelled. It should be noted that the precision and complexity of TouringMachines' space-time projection functions can be chosen by the agents' designer at will: functions are provided by the user through suitable **SpaceTimeProjection** declarations when creating the particular TouringWorld environment to be investigated (see Chapter 7). It is worth stressing also that the point of providing these functions is not so much to endow TouringMachines with fully accurate temporal reasoning capabilities, but rather, to allow them to take into account some extra information — entities' intentions — when constructing projections. The belief here is that this extra information could potentially make them more predictive than if they were to rely on inertial projections alone.[12] Now, once in possession of the projected paths for each modelled entity, the agent can proceed to search for a number of different intra- and inter-agent

[12]Indeed, the experiment reported in Section 8.3.1 certainly lends some weight this to supposition.

goal conflicts which might yet occur during the projected time period.

6.5.4 Handling Goal Conflicts

Motivated, as mentioned above, by the need to satisfy each one of its goals, a TouringMachine, through its Goal Conflict Detector, will attempt to identify any events which might threaten or conflict with its goals and, where necessary, will take action to resolve these conflicts by carrying out suitable action recovery procedures. To identify goal conflicts, a TouringMachine makes use of the contents of its *Conflict Library* (see Figure 6.8). The Conflict Library contains a collection of domain-dependent *conflict resolution triples* or *rules* of the form:

$$\langle Conflict, Goal, Resolution \rangle$$

each of which permits the agent to associate any specific instance of a threat or $Conflict$ with a particular desire or $Goal$ that it threatens, and with a recommended recovery procedure or $Resolution$ which should be followed if the agent's goal is to be protected. For instance, in the TouringWorld domain, a rule exists which associates a collision-type conflict (which might occur, for example, when two agents simultaneously approach an uncontrolled junction), with the particular goal affected (in this case, `avoid-collisions`), and with a specific recovery procedure which should be followed (for example, adopting the intention `stop-at-junction`). These rules are supplied to the Library by the user through suitable **ConflictResolutionRule** declarations (see Chapter 7 and Appendix A). More on this below.

Several different types of conflict can be identified by an agent's Goal Conflict Detector in the TouringWorld domain, a number of which have already been referred to above. In particular, conflicts are identified: *(i)* when the agent attempts to execute some action that causes some physical capability conflict (for example, when trying to exceed its maximum speed or when the agent's Planner attempts to retrieve a schema which does not reside in the Schema Library) — the Goal Conflict Detector is made aware of these conflicts through reception of appropriate warning messages from the Prediction Module's Message Handler (see Section 6.5.2); *(ii)* when the agent detects discrepancies in its self model between its actual and expected physical configuration — the Goal Conflict Detector is made aware of these upon accessing appropriate records which were earlier created by the Explanation Module's Model Discrepancy Handler (see Section 6.4.3); and *(iii)* when, after having taken the opportunity in the Explanation stage to check whether any of its own goals were under threat, the agent identified a possible problem — the Goal Conflict Detector is made aware of these upon accessing appropriate records which, also, were created by the Explanation Module's Model Discrepancy Handler (see Section 6.4.3).

A fourth type of conflict which has yet to be described is that which can result when the projected paths of two or more entities are seen to intersect in space-time. In the TouringWorld domain such conflicts are called *collisions*. Having computed, as described above, a projected path for every modelled world entity (including one for itself), the agent's Goal Conflict Detector can identify any potential collisions by performing pair-wise intersections of all of these space-time trajectories and then selecting any "valid" collision points which are seen to occur. Valid collision points are those which occur at some point inside the time period established by the agent's computed spatio-temporal projections.[13] It is important to note here that since an agent's Goal Conflict Detector can intersect the projected paths of any two entities which are currently being modelled, it becomes possible for the agent to detect potential conflicts in *other* entities' goal sets, *as well as* in its own.

Now, in any given timeslice, an entity's goals may be — and, more often than not, will be — involved in a number of identified conflicts. For example, in addition to noticing a decrease in its desired travel speed, an agent might also have projected that it will shortly hit some obstacle if it continues along its present trajectory; here, two goals of the agent are likely being threatened: `reach-destination` (assuming it has a suitably constraining `travel-time` argument) and `avoid-collisions`, respectively. In cases where multiple goal conflicts are identified, the particular one that the agent will choose to resolve will ultimately depend on a number of factors. First and foremost will be the *priority* of each goal under threat: the goals in an agent's **desires** set are prioritised in some domain-specific manner and this ordering should be observed when choosing among multiple conflicts. The goal ordering used in all examples in this dissertation is, from highest to lowest priority, `avoid-collisions`, `obey-regulations`, and `reach-destination`. A second factor to consider will be the *space-time urgency* of the conflict: if both a low and a high priority goal are being threatened and the conflict involving the low priority goal occurs earlier in space-time, it will often be advisable for the agent to resolve the lower priority one first. An example illustrating this case will be given below.

A third factor to consider will be the *environmental constraints* affecting the agent's possible behaviours: in particular, the presence of other obstacles

[13]The Goal Conflict Detector uses precisely the same quadratic motion projection and collision detection functions which are employed by WorldUpdater, the TouringWorld Testbed process responsible for creating plausible simulations of TouringMachines' physical actions. Details of these functions will be deferred until Section 7.4.2. Note also that since entities' projected paths are represented as lists of (possibly) multiple sub-trajectories, path intersections will have to be performed one spatio-temporal sub-trajectory at a time.

or entities may constrain the possible actions that the agent can take to resolve its goal conflicts. For example, when an agent travelling along a two-lane path approaches, in the same lane, a slower moving agent from behind, it will eventually (upon projecting the two motion trajectories involved) detect a threat to its `avoid-collisions` goal. In deciding how to resolve this conflict, the agent will likely be faced with a choice: overtake the slower agent in front of it or slow down to match the other agent's speed. Typically, the former choice would be preferable since the latter, while adequate for resolving the agent's original `avoid-collisions` goal conflict, will, if the agent is operating under any temporal constraints, almost certainly trigger a subsequent `reach-destination` conflict. On the other hand, while overtaking may be preferable, it may not always be possible: for instance, the passing lane into which the agent must move may presently be blocked by some obstruction.

A fourth factor to consider will be the set of conventions or *rights-of-way* which are in force and are accepted as common knowledge or beliefs in the agent's domain. In a world where agents have similar goals and beliefs, predicting how other agents will resolve particular goal conflicts can often lead to an infinite regress as each agent considers the effect it might be having on other agents' decisions about how any other agents currently affect their conflict resolution choices. Rights-of-way are necessary to ensure that such predictions eventually "bottom out". In addition, they can also have the effect of "cancelling" certain other goal conflicts: in the overtaking example above, the slower moving agent will also detect its own `avoid-collisions` conflict (both agents would be affected by the crash), but, upon consideration of the rights-of-way that are in force in the domain, it would simply choose not to take any action.

Similar in aim to the contextual analysis of behaviour performed by Wood's Plan Recogniser [Woo90], consideration of all of the above factors, combined, enables an agent to decide which, among all possible behavioural outcomes identified for each entity in its model set, are the most likely responses that can be expected given the particular set of intra- and inter-agent goal conflicts which have been identified.[14] The same factors, it should be noted, are involved when an agent, having detected one or more conflicts in the goal set of another entity, is faced with deciding which of the identified conflicts *the entity itself* will choose to resolve. Here, the agent must make certain assumptions of similitude: in particular, that the entity has similar goals

[14] Wood [Woo90], in fact, concentrates primarily on the analysis of environmental constraints such as physical (agent-object, agent-agent) situational constraints and legal driving speeds; her Plan Recogniser appears to pay less attention to either the motivational force or the constraining influence that an agent's (prioritised) goals can exercise on the agent's own — and, for that matter, other entities' — choices of action.

and goal priorities, that the entity will have identified the same set of goal conflicts and environmental constraints, that the entity has the same beliefs regarding rights-of-way, and that the entity is rationally committed, just as the agent itself would be, to resolving any conflicts which it knew might affect it. In terms of simulating other entities at the knowledge level (this notion was introduced at the start of Section 6.5.3), Newell [New82, pages 101–105] refers to this as applying an *extended principle of rationality*. This states that, given certain requisite initial and boundary conditions of some entity — its goals, intentions, and initial and acquired knowledge or beliefs — together with knowledge regarding the entity's goal preferences and the physical plausibility of its different action choices, an agent, under certain conditions, will be able to calculate the trajectory of the modelled entity.

Having performed, by now, all necessary projections of the entities in its model set and having subsequently performed, via application of its prioritised conflict resolution rules, an appropriate contextual analysis of these entities' current and predicted behaviours, the agent will be in a position to determine, for each entity, which goal conflict among all those identified is the most pressing and, in response to each conflict, which specific resolution method will be adopted to counter it. Conflict resolution methods can solicit one of two response types from the entity concerned: either the entity will need to adopt a new intention — for example, `stop-at-junction` or `start-overtake` — or, it will need to effect some primitive physical action — for example, `change-speed` or `change-orientation`.

The agent will consider adopting a new intention in situations where it decides that its own current intention (or the intention it currently ascribes to some other entity) is causing itself (or the other entity) to behave in a manner which gives rise to a conflict: for example, going through a junction without having right of way, going through a red light without stopping, or hitting another slower moving agent from behind. In the case where it is the agent itself which must adopt a new intention, the agent's Goal Conflict Detector will send an appropriate task (intention) command message to the agent's Planner in layer $\mathcal{P}$ (more on this below). Where it turns out to be some other entity which is expected to adopt a new intention, the agent will simply make a suitable alteration to the **intentions** component of its model of the entity, thereby reflecting the new task that the entity can now be expected to plan for. This type of conflict resolution procedure, incidentally, can be regarded as a very simple form of plan modification or replanning. More sophisticated implementations of this sort of procedure have been proposed by a number of researchers, including Wilkins [Wil85], Ambros-Ingerson and Steel [AIS90], and Wood [Woo90].

Unlike intention changes, primitive physical action responses will only be considered when the modelling agent *itself* has become involved in a conflict of the type identified by a *self model* discrepancy (for example, as a result of a loss of speed or directional deviation from its intended target). Described earlier in this section, and also in Section 6.4.3, these conflicts come about, not because the agent's current intention is at fault, but because a simple discrepancy has occurred between the agent's desired and actual configurations. For such conflicts, then, simple "self-tuning" actions can be considered appropriate.

Before giving examples illustrating the various prediction processes that have been detailed above, it is necessary at this point to describe one remaining and very important operational characteristic of the Prediction Module. The description thus far has ignored the fact that TouringMachines, when making predictions about entities' future trajectories and conflict resolution behaviours, are capable, at the same time, of carrying out a simple but also very powerful form of hypothetical or *counterfactual reasoning*. In particular, TouringMachines are able, for a given number of counterfactual levels — this number is set for the agent by the user via the Testbed parameter **ConflictResolutionDepth** (see Chapter 7) — to project and reason about any modelled entities' behaviours while also, at each level of reasoning, taking into account any actions resulting from the entities' earlier attempts to resolve their goal conflicts. In other words, when constructing entity model projections at some counterfactual reasoning level N, say, the agent will be able to take into account any conflicts *plus any actions resulting from the anticipated resolutions to these conflicts* which it had previously detected at level $N - 1$. Thus, by setting the parameter **ConflictResolutionDepth** to any value greater than 1, an agent will have the ability to take into account (up to the given number of nested levels of modelling) any entity's responses to any other entity's responses to any predicted conflicts. The potential impact that this kind of reasoning can have on an agent's ability to perform timely and effective predictions will be analysed more closely in an experiment in Chapter 8. Examples of agents' general modelling capabilities now follow.

A TouringMachine, as mentioned in the previous chapter, initially sets out to achieve its goals without any prior knowledge of other agents whereabouts — indeed, without even knowing whether other agents exist. As such, when approaching an uncontrolled junction at which, it has been established, it will have to alter direction, a TouringMachine will not plan — in advance — to give way to other agents. Rather, it will simply plan to stop in the middle of the junction, turn to the appropriate direction, and then proceed toward its destination. In the scenario of Figure 6.9, two agents, a round one (`agent1`) and a chevron-shaped one (`agent2`), can be seen approaching an uncontrolled junction at time $T = 9.0$ (upper left-hand frame). At time $T = 11.0$

(upper right-hand frame), the two agents sense and construct models of each other for the first time (`agent2`'s sensing arc is displayed here for illustrative purposes only). When the agents subsequently project each other's expected trajectories, they each will notice an impending `avoid-collisions` conflict. However, because of their common beliefs about rights-of-way, `agent1` will predict that `agent2` will resolve the conflict by subsequently taking on the intention `stop-at-junction`. Conversely, `agent2` will predict, correctly, that `agent1` will continue on its present trajectory through the intersection (see lower left-hand frame of Figure 6.9). Only when `agent1` leaves the junction will `agent2` (which has been polling `agent1`'s progress through the junction ever since it had to stop around $T = 15.0$) drop its adopted conflict resolution intention and proceed into the junction and on with its original task (lower right-hand frame).

A slightly different conflict situation is illustrated in Figure 6.10. In this scenario, two agents, a chevron-shaped one (`agent1`) and a round one (`agent2`), can be seen approaching a light-controlled junction at time $T = 6.0$ (upper left-hand frame). At time $T = 8.0$ (upper right-hand frame), when the two agents sense and construct models of each other for the first time, a number of events take place. Like the agents in the previous example, these, upon projection of their respective trajectories, will predict an impending collision involving each other; in other words, a mutual `avoid-collisions` conflict. However, at the same time, another event occurs which triggers an additional conflict in `agent1`'s goal set: the traffic light which `agent1` is approaching has turned from green to amber.[15] Here, it is `agent1`'s `obey-regulations` goal which is under threat since running through red or amber lights is not permitted in the TouringWorld domain. Now, although `obey-regulations` will generally be considered — certainly when viewed in terms of a TouringMachine's static goal preferences in its **desires** model component — to be of lower priority than `avoid-collisions` (`agent1`'s other goal currently under threat), `agent1`, and `agent2` for that matter, will, upon application of their respective conflict resolution rules, ultimately identify the conflict involving `agent1` and the traffic light to be more constraining in the current situational context. As a result, `agent1` will proceed, as required and predicted, to carry out a suitable resolution procedure which protects its `obey-regulations` goal: in this case, through adopting the intention `stop-at-light` (Figure 6.10, lower left-hand frame). Only when `agent1`'s light changes to green (lower right-hand frame)

---

[15] A traffic light agent is represented graphically in the TouringWorld as a rectangle divided into two squares: when a black circle appears only in the bottom of the two squares, the light is communicating the colour `green`; when a circle appears only in the top square, the light is at `red`; when both squares display black circles, the light is at `amber`. The terms "bottom" (closest) and "top" (farthest) are relative to the agent to which the light applies.

```
process pending input messages
foreach Entity ∈ M_T do CRM_Entity := nil
foreach Entity ∈ M_T do begin
   N := ConflictResolutionDepth
   for CounterfactualLevel := N downto 1 do begin
      project anticipated space-time trajectory by considering CRM_Entity
      determine intersections with other entities' trajectories
      if Entity involved in any type(s) of goal conflict(s)
      then {determine most pressing goal conflict,
               CRM_Entity := most contextually appropriate conflict resolution method}
      else CRM_Entity := nil
   end
end
foreach Entity ∈ M_T do begin
   project anticipated space-time trajectory by considering CRM_Entity
   create and store expectations in M_T for use later in timeslice T + 1
end
if Entity = self and CRM_Entity ≠ nil
then submit for processing the action or intention CRM_Entity
```

Figure 6.11: Algorithm to perform model-based predictions of entity behaviour (pseudocode). $\mathcal{M}_T$ is the agent's collection of stored entity models at time T.

will it proceed toward its original target destination. Note, also, that as a result of its unplanned stop, `agent1` might subsequently detect a conflict with its (presumably) time-constrained `reach-destination` goal. Were such a conflict to arise, `agent1` would likely take to resolving this with a suitably gauged `change-speed` action.

6.5.5 Generating Expectations and Closing the Loop

Once all future conflicts have been detected and their respective resolutions duly identified, two operations remain to be carried out by the Prediction Module. The first of these is to generate the expected configuration of each entity in the agent's model set. This is carried out by making one last projection of each entity's anticipated trajectory, taking into account any final conflict resolution action or intention change which the agent expects the entity to carry out. By considering the projection time period to be the interval between the agent's successive model updating operations,[16] each entity's expected configuration will be computed and then stored in the **expectations** component of the agent's respective entity model.

The expectations of a modelled entity, it should be noted, only reflect the entity's expected physical configuration. An implicit assumption being made in this model updating process, then, is that the entity's other modelled mental states (its beliefs, desires, and intentions) will, at least at this stage of processing, be assumed correct and so will persist from one timeslice to the next — until there is explicit evidence that these mental states of the entity have changed. (How and when modelled intentions get revised has already been addressed above.) In the TouringWorld domain and, more specifically, in TouringWorld environments populated with homogeneous agents such as those considered in this dissertation, this assumption is a reasonable one to make. On the other hand, in richer, less constrained environments, agents might be required to represent substantially more complex beliefs about their changing surroundings, necessitating as a result, suitably powerful belief maintenance or revision capabilities. This and related issues (for example, agents' commitment to their goals) are re-visited in Chapter 9.

The last step taken by the Prediction Module will be to process any conflict resolution proposal which has been determined by the Goal Conflict Detector and which applies to the agent itself. In particular, where the proposed conflict resolution is a primitive action, an appropriate action command will be sent to the agent's effectors for subsequent consideration. Where the proposed resolution is a suggested intention change, an appropriate command message will be sent to the agent's Planner in layer $\mathcal{P}$ — the process of handling such messages was described in Section 5.4.2. With this, then, a full cycle of a TouringMachine's modelling functions comes to an end. To conclude this subsection, the full pseudocode algorithm implementing the various processes of the Prediction Module is given in Figure 6.11.

6.6 Modelling in the TouringWorld

A TouringMachine's modelling layer, $\mathcal{M}$, is charged with constructing causal models of agent behaviour. Models are used by a TouringMachine in order to monitor, explain, and predict the behaviours exhibited by the different agents which appear in its model set. Through the use of a Library of model templates, the TouringMachine is able to assign, resources permitting, default models to any agents which it has encountered in the environment. As subsequent

[16]This is established with the Testbed parameter **ModellingRate** — see Chapter 7.

observations of these agents are made and discrepancies in their respective models detected, an abductive reasoning process is invoked in order to explain these discrepancies. Having settled on suitable theories which explain the observed behaviours, the models can then be used for generating predictions of potential intra- and inter-agent goal conflicts and for creating expectations of future agent behaviours.

In addition to modelling itself and other mobile agents like itself, a TouringMachine agent can build models of other TouringWorld entity types — agents and objects alike — so long as suitable templates are made available in the agent's Model Library. The purpose and usefulness of modelling non-TouringMachine entities can be illustrated, for instance, by considering the scenario described above in Figure 6.10. In this scenario, an agent is able to detect and reason about such conflicts as running through red traffic lights only if it possesses models of the traffic lights in its environment. In particular, the agent will be able to detect pertinent `obey-regulations` goal conflicts (like the one which takes place in this scenario) by projecting the configuration of whichever entity it is currently modelling and then intersecting the resulting space-time trajectory with the projected configuration of the appropriate traffic light.[17] Here, two types of possible collision-type conflicts must be distinguished. In addition to the *physical* collision which would occur if the modelled entity were projected to hit the traffic light concerned (in this case, the entity's goal under threat would be `avoid-collisions`), there is also the possibility of a *virtual* collision which would occur if the entity were projected to pass over the traffic light's stop line (these lines are not graphically illustrated in any of the figures in this dissertation) at the same time that the light is communicating the colours `red` or `amber`. Unless traffic lights are modelled by the agents concerned, such conflicts will go undetected.

Besides modelling immobile agents like traffic lights, a TouringMachine can also, if necessary, maintain models of immobile objects such as obstacles, kerbs, walls, and even lane markings (other static entity types will be described in the next chapter). Being inert and lacking "interesting" mental states of any sort, these objects can be modelled and reasoned about fairly trivially (and, not surprisingly, with a very high degree of predictive accuracy). As described in Chapter 4, a TouringMachine can already respond to unexpected interactions with such immobile objects by making use of various hardwired situation-action rules which reside in its reactive layer $\mathcal{R}$. Because of their (extremely) simple behaviour, objects, it would seem, are probably able to be handled more than adequately by the agent's reactive layer. The only

[17] A traffic light's configuration is calculated as if the light were a (permanently) stationary TouringMachine.

obvious advantage of allowing a TouringMachine to model such static entities is that it could enable the agent to construct earlier and, therefore, possibly more reasoned accounts of the conflicts to which these entities give rise. This conjecture, however, has not been explored in this dissertation.

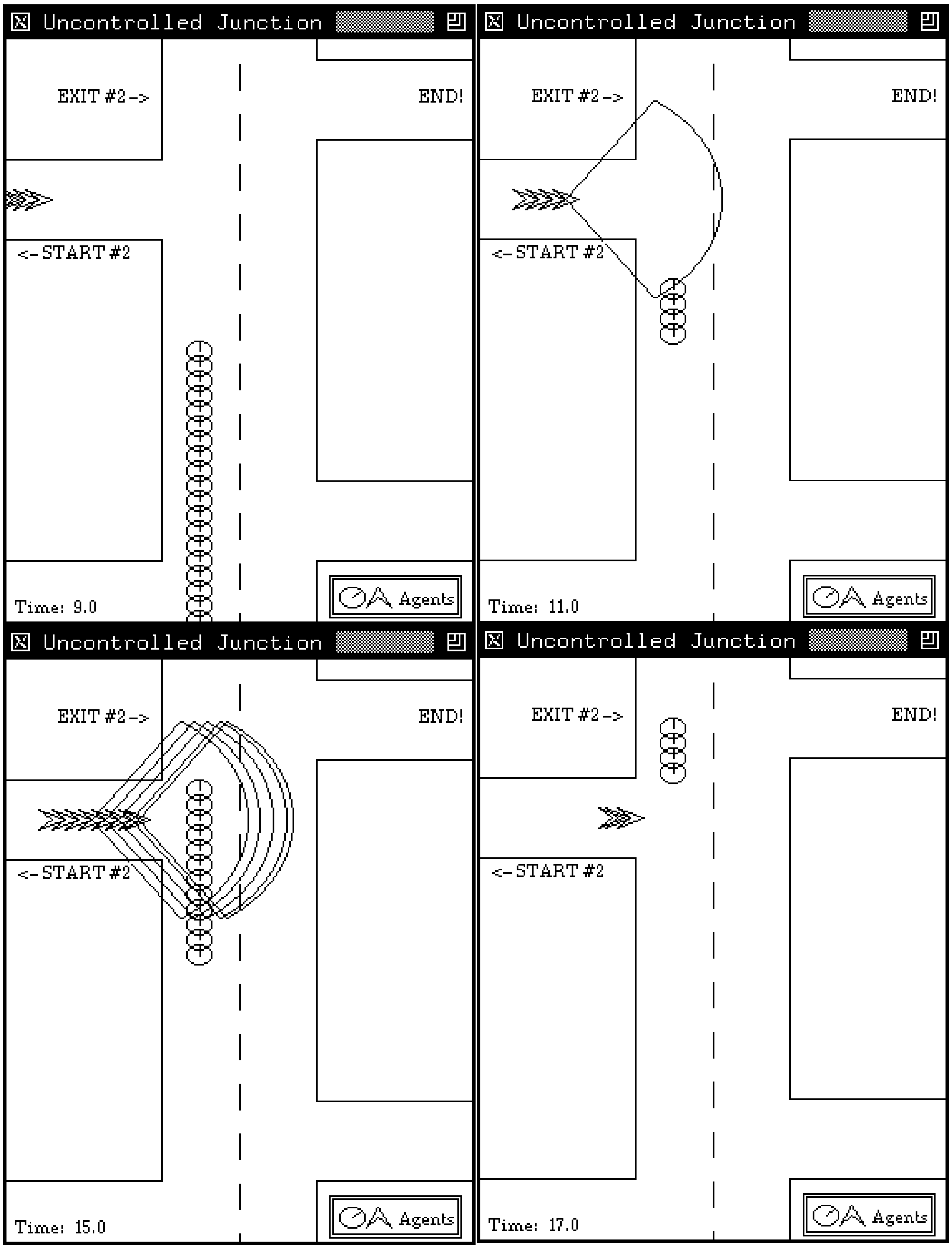

Figure 6.9: Resolving goal conflicts at an uncontrolled junction.

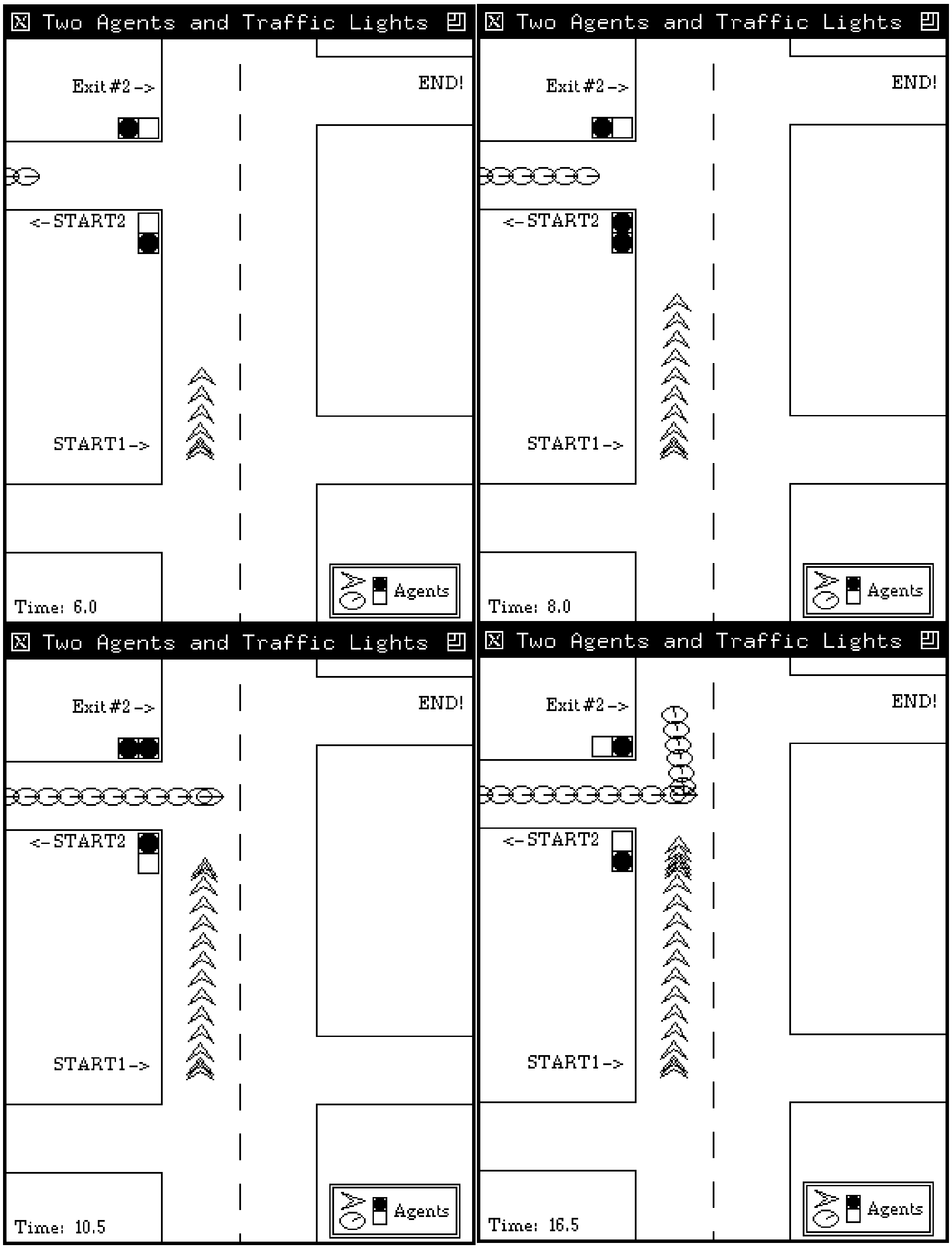

Figure 6.10: Resolving goal conflicts at a light-controlled junction.

7

The TouringWorld Testbed

Control — that's the proof of civilization. Anyone can do something once, but repeating it and maintaining it — that's the true test.

Paul Theroux, *The Mosquito Coast*

7.1 Introduction

In contrast with much of the early work on AI planning systems, there has been a growing belief amongst researchers involved in agent architecture design that the appropriateness of different agent designs or configurations (that is, their particular skills and capabilities) very much depends on the characteristics of the environments within which the agents are intended to operate [CGHH89, DM89, HHC90, PR90, KG91].

TouringMachines are intended for use in fairly complex domains. In particular, domains that are dynamic (events occur which are beyond the control of individual agents), unpredictable (agents are neither omniscient nor prescient), real-time (the pace of world change is a reality and agents must respond to it within bounded time limits), and ongoing (there is no single, well-defined problem to be solved). Because TouringMachines are capable of integrating a diverse range of reactive and deliberative behaviours, one would expect, at least in principle, that TouringMachines should be able to carry out with success the types of tasks for which they are intended.

TouringMachines, however, are fairly complex and sophisticated machines, rich with different skills and capabilities. And since they operate in rather complex environments, the precise relationship between a TouringMachine's particular configuration, the behaviours it exhibits, and the environmental conditions that might influence these behaviours is not, at least at the outset, entirely clear. If a TouringMachine's performance is to be robust and, ultimately, predictable across a range of different task-domains, a sound understanding of the agent's behavioural ecology — the relationship between the agent's structure or configuration, its environment, and its resulting behaviours — would seem crucial. Attempting to define an agent's behavioural ecology — a term borrowed from Cohen *et al.* [CGHH89] — is that of building an abstract causal model which describes the relationship between the agent's performance and its environment. The function of such a model, then, would be to enable, through empirical investigation, both *explanation* of an agent's observed behaviours, as well as testing or *prediction* of its hypothesised behaviours.

In order to gain a fuller understanding of the behavioural ecology of TouringMachines, a feature-rich multi-agent testbed — the *TouringWorld Testbed* — has been designed and implemented for use with TouringMachines. The Testbed provides a platform for performing empirical investigations of agent behaviour under a wide range of user-controlled environmental conditions. One area of particular interest, for instance, is understanding the influence different environmental conditions might have on a TouringMachine's ability to coordinate its activity with other agents while at the same time trying to accomplish whichever initial tasks were assigned to it. The results of some preliminary investigations using the TouringWorld Testbed are presented in the next chapter.

7.2 Overview of the TouringWorld Testbed

The Testbed (see Figure 7.1) is an instrumented system for building and analysing simulated environments inhabited by one or more task-achieving TouringMachines. The Testbed is centred around a deterministic, discrete event simulator which realistically mimics the environmental dynamics of the TouringWorld — the particular multi-agent domain chosen for studying TouringMachines. While not based on any specific real-world application, the TouringWorld can be regarded as a reasonably faithful approximation of a particular class of domains: those which are partially-structured and comprise a number of agents acting in real time (for example, an automated factory floor or traffic environment).[1] It is important to remember, however, that since the

[1] The choice of domain was arrived at through participation in the discussions on Motorway Vehicle Rationality held at SRI Cambridge during April and May, 1989. I am particularly indebted to Ben Macías, Ann Copestake, John Levine, Steve Pulman, and Julia Galliers for their input and assistance on this matter.

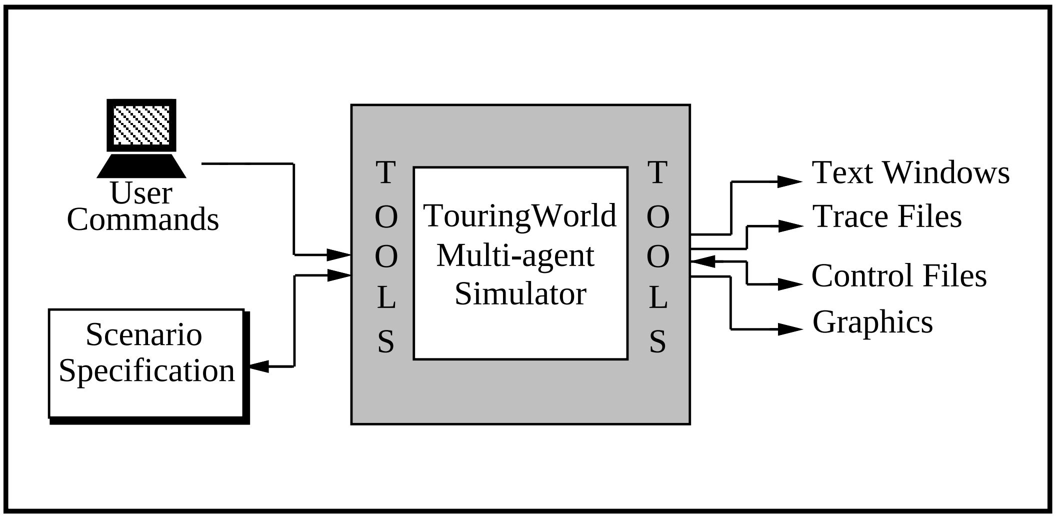

Figure 7.1: Top-level view of the TouringWorld Multi-agent Testbed.

focus of interest here in on issues pertaining to *agent-level* performance and on how *agents* can coordinate their tasks in the presence of other agents, certain aspects of such real-world domains have been appropriately abstracted, primarily to minimise the processing complexity of simulation, but also to reduce the knowledge engineering effort involved in describing the domains. The TouringWorld domain will be described in more detail below.

Along with a multi-agent simulator, the Testbed also comprises a series of user-level tools with which to define, view, control, record, alter, and analyse the behaviours or characteristics of the TouringMachines under study. Besides allowing the user to change various features or capabilities within the agents themselves — for example, their assigned tasks or their per-timeslice resource limits — the tools also facilitate altering various characteristics of the agents' environment — for example, the density and type of obstacles or the rate of passing of time — and thus enable the user to study a range of agent configurations under a wide range of different environmental conditions.

A user can interact with the Testbed via a range of different I/O facilities, including window- and line-oriented textual displays of selected scenario information, a graphical display of changing scenario events, plotted graphs of various agent and environmental statistics, an interactive function for "driving" a user-controlled agent, as well as file and window output of tracing information chronicling the events that took place during the running of the scenario.

In order to permit extensive experimentation, TouringWorld agents and environments have, from the earliest stages of their design, been implemented so as to be customisable by the user by means of a rich collection of entity- and environment-level *parameters*. A parametrized Testbed facilitates investigations aimed at understanding the behavioural ecology of TouringMachines by making it relatively fast and easy for the user to experiment with different agent and environment descriptions. This decision to parametrize the Testbed also reflects the opinion shared with a growing number of researchers in DAI and *Distributed Problem Solving* (DPS) that extensive experimentation, if not initially conceived and developed as a primary goal of the system design, will be very difficult to carry out once the system is complete [LC83, GBH87, DM89].

The user supplies specific values for entity- (agent/object) and environment-level parameters via a *Scenario Specification* file (see Figure 7.1). Parameter definitions are expressed in a declarative language called the *TouringWorld Scenario Specification Language*, or *SSL* (a complete extended BNF grammar is given in Appendix A). Besides containing all of the initial data (parameter values) describing the particular TouringWorld environment and entity set to be studied, the Scenario Specification also contains a number of scenario-level SSL statements which define, among other things, how many times the scenario should be iterated, which parameter values should be automatically changed between iterations, and where on the user's terminal screen each of the various I/O windows should appear. Each of the various scenario-, environment-, and entity-level TouringWorld parameters will be described in Section 7.4 below.

Before proceeding to give more details about the TouringWorld domain and Testbed, it is worth mentioning why an empirical approach to studying TouringMachines was favoured over a formal analytical one. Essentially, it is the author's opinion that despite the fairly recent proliferation of different proposals for integrated agent architectures, this new AI subfield is very much in its infancy and as such does not lend itself easily to formal analysis. What pertinent theoretical work there is — for example, that of Cohen and Levesque on formalising the role of intentions in agents [CL87] or that of Bratman *et al.* examining the trade-off between reaction and deliberation [BIP88] — is, as Kinny and Georgeff have rightly argued [KG91], very general and says little about specific real-time reasoning strategies and their effect on agent behaviour. This is also true of other theoretical work, including that of Dean and Boddy on time-dependent planning [DB86] and that of Russell and Wefald on utility-based deliberation [RW89].

In the absence of a comprehensive theory explaining the behaviour of real-time computational agents, use of a controlled environment — a parametrized *artificial* world — is, at present, the only practical avenue for measuring and studying agent performance.[2] Emphasizing the relatively immature state of present research in their respective subfields, similar arguments have been made by researchers in DPS systems [LC83, DM89], DAI systems [GBH87], planning systems [LD90], and also machine learning systems [Lan88].

It is also interesting that many researchers, having noticing the rather urgent need for a set of common benchmark tasks and environments with which to compare different agent architectures, have increasingly started to promote the use of simulators which, by providing precise computational descriptions of different task environments, facilitate the direct comparison of different architectures [CGHH89, DK90, PR90] and favour the possibility of transfer of techniques between different applications [GBH87, DM89].

7.3 The TouringWorld Domain

The TouringWorld is a dynamic, ongoing real-time domain populated by any number of *entities*. An entity can either be of type *object* or of type *agent*. Objects are static structures which are either *collidable* (for example, obstacle, wall, kerb, information-sign) or *non-collidable* (for example, entry, exit, lane marking). Agents, on the other hand, are dynamic, task-achieving entities which are either *mobile* (that is, TouringMachines) or of type *environment* (for example, traffic-light, rain, fog). A number of these entity types are illustrated in Figure 7.2. While objects always remain at fixed initial locations throughout an entire scenario, agents, through the execution of assigned tasks, are able to change either their location (TouringMachines) or their state (TouringMachines and environment agents) or both. In order to standardise the treatment of all agents, both agent types — mobile and environment — are assigned initial tasks which they carry out by building and executing plans. Traffic lights, for example, are assigned the task `light` which causes them to repeatedly change their colour (`red`, `green`, or `amber`) at a given rate.[3] Likewise, if the fog agent is present in the scenario, it is "assigned" the task `fog` which causes it to change, for a specified time period, the value of a scenario parameter which is used by the simulator to determine a TouringMachine's field and range of vision. The rain agent has a similar effect on another scenario

[2] Since the emphasis of this thesis is on the design of cognitive-level agent processes rather than on such functions as low-level navigation, real-world testing was ruled out as a viable platform for studying TouringMachines. Simulations are not a perfect substitute for the real thing and should ideally be done in conjunction with real-world testing. For the time being, however, this must remain a potential avenue for future research.

[3] The precise argument lists used in the specification of environment agents' tasks are defined in Appendix A, page 191, under the syntax rule for *planner-task*.

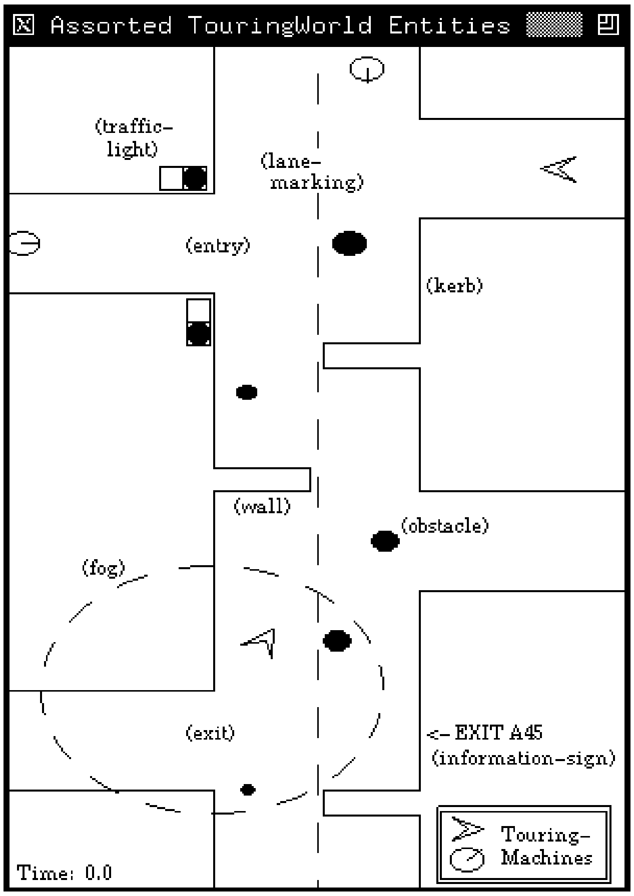

Figure 7.2: A snapshot of a particular TouringWorld scenario showing various types of (labelled) objects and agents.

parameter which is used by the simulator to calculate a TouringMachine's braking distance (see below).

Mobile agents are TouringMachines: in other words, autonomous route-planning vehicles which attempt, within some given deadline, to relocate to one location or another as specified by the arguments to their respective `plan-a-route` tasks. In addition to this single achievement goal, each TouringMachine also has homeostatic goals to avoid colliding with other entities and to obey various traffic regulations.

Each TouringMachine starts out with some geographical knowledge of the world — its Topological Database contains locations of paths, path junctions, and certain landmarks associated with these junctions — but has no prior knowledge of the whereabouts of other agents or of any obstacles. Because TouringMachines move around independently from one another in real time, with no specific knowledge regarding each other's ultimate destinations, their paths will invariably cross: that is, TouringMachines will invariably enter into states of conflict with one another. Additionally, because each agent is resource-bounded and has only limited capabilities with which to sense and monitor the world, TouringMachines will only ever have a limited time to take actions and so may well, from time to time, make errors of judgement, some resulting in terminal collisions.

Many aspects of the TouringWorld pose considerable challenges to agents: in particular, TouringMachines have limited computational resources, limited physical and reasoning capabilities, upper bounds on their speed, acceleration and steering rate, and limited knowledge of other entities' tasks or whereabouts. While it would be fair, then, to describe the TouringWorld as a reasonably faithful approximation to certain types of real-world, multi-agent domains (for example, highly automated factory floors or traffic environments) a number of simplifications were nevertheless made, both to reduce the processing complexity of the Testbed and to minimise the knowledge engineering effort involved in simulating realistic task-domains. Specifically, the major simplifications made in the TouringWorld Testbed include:

- The environment is represented as a $2\frac{1}{2}$-dimensional grid with a parametrized spatial resolution whose value is chosen by the user. (The extra $\frac{1}{2}$ dimension comes from the fact that certain entities such as walls, obstacles, and agents have a nominal height component and so can — from the point of view of an agent sensing the environment — occlude some other smaller or more distant entity.)

- To simplify both the calculation of agents' simulated perception fields and the computation of contact points when entities collide with each other, TouringMachines and certain other entity types — obstacles, information signs, and environment agents — are modelled as circles. Additionally, collidable entities such as TouringMachines and obstacles are modelled as being solid but massless. This means that if two such entities collide they will come to a complete halt at the first point of contact. Also with the aim of simplifying the simulator's computational load, all remaining TouringWorld entity types — traffic lights, kerbs, walls, paths, path entries and exits, and lane markings — must be rectilinear and placed orthogonally with respect to the simulated world's global orientation system (right-hand side of screen window $= 0°$, top $= 90°$, left-hand side $= 180°$, and bottom $= 270°$).

- Collisions always occur whenever two or more collidable entities attempt to occupy the same physical space at the same time. In particular, then, the Testbed does not implement other forms of collision handling which might possibly be of some use in certain predator-prey environments (for example, entities becoming linked together or entities bouncing off one another — see Durfee and Montgomery [DM89]).

- A TouringMachine is a car-like mobile robot which moves with three degrees of freedom: translational displacement across the two major dimensions of space, x and y, plus rotation θ about its centroid. Being "car-like", a TouringMachine is subject to so-called *nonholonomic* kinematic constraints: it cannot move sideways and its turning radius is lower-bounded [Lat91]. To simplify the calculations that agents need to perform when planning motion trajectories, all paths and path junctions have been made rectilinear. To turn at junctions, agents must come to a complete stop before changing orientation. Path surfaces are assumed to be smooth so issues such as wheel slippage have been ignored.

- TouringMachines sense and act on the environment at discrete time intervals. In particular, if a TouringMachine effects an action during a particular interval or timeslice, the action — or some suitable portion of it — is assumed to occur instantaneously at the end of the timeslice.[4] Timeslice size is defined as a scenario parameter whose value is chosen by the user (see below).

- Through sensing of their environment, TouringMachines are able to identify (without any ambiguity or error) other entities' *physical* properties: that is, their Cartesian locations, speeds, acceleration rates, orientations, and communicated information (for example, brake lights or indicators). Agents always believe what they see — more precisely, they believe what their sensors and Focus of Attention mechanisms happen to have registered at the start of a given timeslice. An entity becomes visible to another agent if any point on that entity's contour happens to fall inside the agent's forward or rear sensing range.[5] TouringMachines are also assumed to be able to distinguish entity uniqueness. Unlike its physical properties, however, an entity's *internal* properties (for example, its beliefs, desires, or intentions) are not immediately recognisable to other agents and so can only be inferred through abduction. All the same, despite any differences in the values of these internal properties, every mobile agent is assumed to be a TouringMachine — with the same basic three-layered control architecture described in Chapters 3 through 6.

[4] As described in Chapter 5, page 81, actions that take more than one timeslice to complete are iterated between the agent's effectors and its Planner.

[5] In fact, if the entity is completely occluded by something like a large obstacle, the agent will only "see" this entity if its sensors are programmed to detect occluded entities. TouringMachines' sensors are "programmed" via the parameter **SensingAlgorithm**, as described in Section 7.4.3 below.

- In terms of resource limitations, TouringMachines are bounded computationally (that is, they can perform up to some parametrized maximum number of processing steps per timeslice) but are assumed to have unlimited consumable resources (for example, fuel).

- TouringMachines communicate using a common, error-free protocol: left and right indicators to turn or overtake, brake lights when braking, and horn, fog lights and flashing headlights to convey specific warnings. Explicit communication plays a very minor role as far as inter-agent coordination is considered, agents relying principally on self-built models for explaining and predicting other agents' behaviours. Were this not the case, the need to consider inter-agent communication errors or delays (as done, for example, in the DVMT [LC83] and MICE [DM89] testbeds) would be more pressing. Note, however, that from the point of view of an agent sensing the world, there is still room for ambiguous interpretation of certain communications. In particular, if one agent observes another agent signalling right it might be that this second agent is about to overtake some third one ahead of it or it might simply be that the second agent is about to turn right onto another path.

- Right-of-way protocols for dealing with most junction scenarios and overtaking situations are assumed common knowledge and are embedded in agents' model template default belief sets. Thus, unlike agents belonging to what could perhaps be described as more general *open systems* (for example, hierarchical organisations), TouringMachines do not have to deal with reconciling disparate viewpoints or with negotiations to resolve conflict situations [Gas91]. Also, agents do not belong to different levels of organisational authority: in MACE terminology, each TouringMachine could be described as an equal *co-worker* [GBH87].

While it is true that TouringMachines are not exercised with tasks that require particularly deep knowledge or detailed domain-specific expertise, the TouringWorld is nevertheless an interesting domain because it challenges a number of commonsense skills in TouringMachines: in particular, their ability to sense and move around in a complex world and carry out time-constrained tasks — always with a degree of uncertainty about what might happen next. Also, since the abilities to perceive and explore the environment are generally regarded as prerequisites for agent autonomy [Mor88, CL91], the TouringWorld domain should prove a suitable domain for evaluating a number of important issues concerning rational autonomous agency.

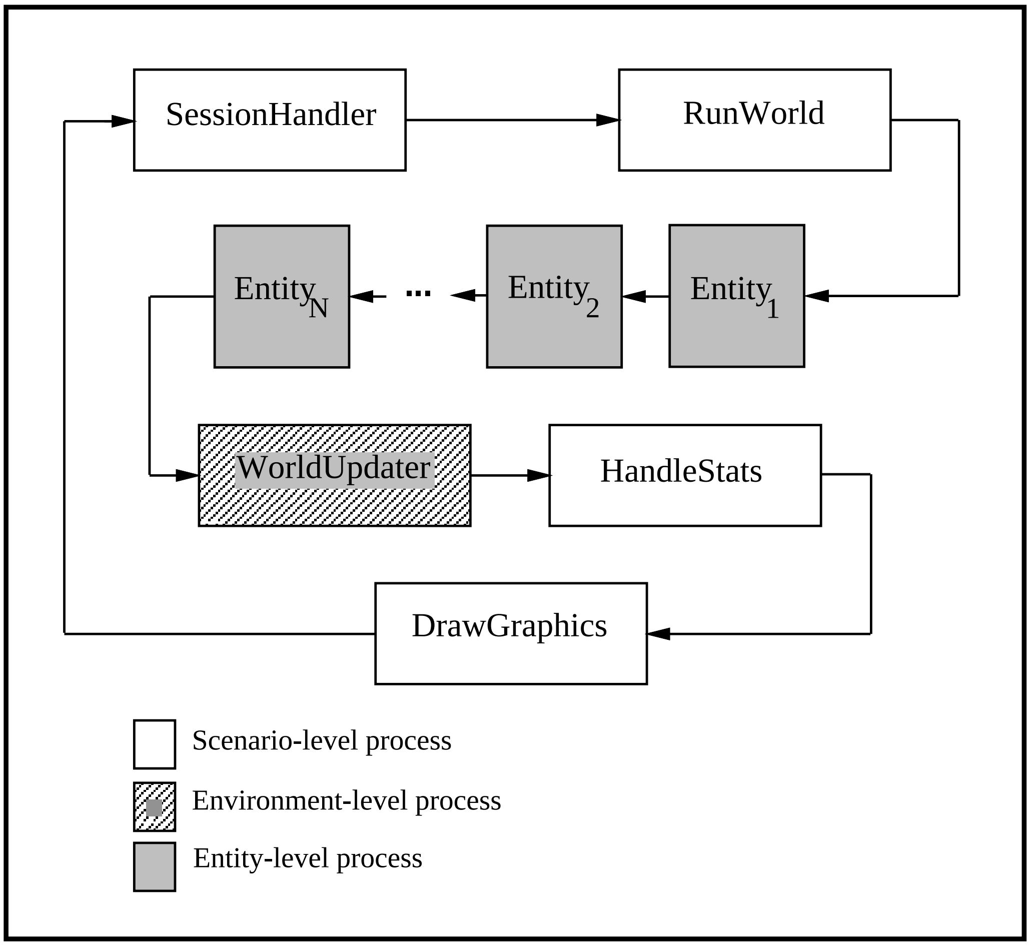

Figure 7.3: The Process Scheduler Queue. The TouringWorld Multi-agent Testbed is implemented as a series of scenario-, environment-, and entity-level processes which are scheduled on a central processing queue and executed in a round-robin fashion.

7.4 The TouringWorld Testbed

As mentioned above and illustrated in Figure 7.1, the main components of the TouringWorld Testbed include a multi-agent simulator and a series of user interface tools for recording, analysing, viewing, and controlling the behaviour of a group of TouringMachines. The aim in this section is to describe some of the details relating to the design and implementation of these two components.

In fact, the various simulator and tool functions are implemented in Prolog[6] as a set of system program modules, or *processes*, which are scheduled on a central processing queue, the *Process Scheduler Queue*, and executed in a round-robin fashion. There are three types of system processes, each im-

[6]SICStus Prolog 0.7 running under SunOS† 4.1. †*SunOS* is a Registered Trademark of SUN Microsystems, Inc.

plementing a mixture of some simulation and some user-interface functions. The first type (see Figure 7.3) are called *scenario-level processes*, and these are responsible for such functions as initialising the Testbed, reading in and processing the Scenario Specification file, figuring out which entities are involved in the scenario, collecting and computing various statistics about the behaviour and performance of agents, as well as administering most of the user-level textual and graphical I/O facilities.[7] The second type comprises the *environment-level process*, which is responsible for providing most of the central simulation functions: collating and processing agents' actions and state changes according to various kinematic and dynamic constraints, updating the simulated world clock, and maintaining an up-to-date *World State* fact base describing entities' most current physical configurations. The third type of processes are called *entity processes*. These, as might be expected, implement the activities performed by the various instances of environment and mobile agents that are present in a TouringWorld scenario.

These different system processes, which collectively implement the TouringWorld Testbed, have been parametrized to enable a high degree of customisation by the user. The parameters used to customise these processes are precisely those which are supplied by the user in Scenario Specification file (see Figure 7.1). As mentioned above, these Testbed parameters are specified by the user in a declarative language called the TouringWorld Scenario Specification Language, or SSL, for which a full extended BNF grammar appears in Appendix A. In order to describe the various parameters, interface tools, and scenario-, environment- and agent-level processes that combine to implement the TouringWorld Testbed, repeated references will be made to this grammar throughout the remaining sections of this chapter.

7.4.1 Scenario-level Processes

7.4.1.1 SessionHandler

The *SessionHandler* process provides the first level of interface between the user and the Testbed. Upon invocation of the TouringWorld Testbed by the user, this process is charged with loading and initialising all of the Prolog program files which collectively implement the Testbed. Having done this, the SessionHandler then prompts the user to select from a menu of stored scenarios the name of the particular scenario to be run.

Each name in the scenario menu is associated with a stored Scenario Spec-

[7] Window and graphics capabilities are provided through the use of XWIP, an interface to the $\mathcal{X}$ Window System for Prolog [Kim90].

ification file which has been created by the user at some earlier date.[8] A Scenario Specification file contains all of the SSL parameter declarations which are used to customise the various scenario-, environment-, and entity-level processes that make up the TouringWorld Testbed. Once the user has selected a scenario name, the SessionHandler reads in the corresponding file, translates its SSL declarations into appropriate Prolog facts and rules, and then loads these alongside the system code as executable Prolog statements.

The SessionHandler is responsible for setting up any I/O channels which may be required by the various scenario-level processes and interface tools. In particular, I/O channels may be needed for displaying certain items of text, for drawing graphics and displaying statistical graph plots, as well as for receiving the user's commands to suspend, restart, or terminate the Testbed's operation. Associated with each of these scenario-level processes is an optional *window-definition* (see Appendix A, page 184 for details of the relevant declarations) which defines, among other things, the dimensions and location of the particular screen window to be used for displaying and receiving each process' I/O. If a scenario process has no associated window definition its I/O will simply be routed to the default screen, ensuring, therefore, that the TouringWorld Testbed remains usable on both line-oriented and window-based terminals. A number of scenario-level windows (`TouringWorld`, `Agent-1`, `Agent-2`, `Graph Plot`, `Actions`, `Scenario Parameters`, and `Two Agents with Traffic Lights`) can be seen in Figure 7.4.

To enable the study of, among other things, the behaviour of a TouringMachine under a *range* of differing environmental conditions, the TouringWorld Testbed provides a built-in facility for iteratively running a scenario (the number of iterations is specified by the **ScenarioIterations** parameter) and for automatically effecting, between iterations, changes to a number of the entity- and environment-level parameter values that appear in the initial Scenario Specification. Parameters to be automatically altered between iterations are specified via **AlterableParameter** declarations which state the name of the parameter to be changed (for example, `world-time-increment`, `max-resources(planner)`, `forward-sensing-horizon`), the scope of the change (which agents the changes should apply to), whether the change should increment or decrement the parameter value, as well as the amount by which the parameter value should be changed. The full set of alterable parameter declarations are defined under *alterable-tmw-parameter* in Appendix A, page 187.

Other parameter declarations pertaining to the SessionHandler process include **ScenarioSuspension** and **TerminationCriterion**. The former is

[8] Scenario Specification files are simply ASCII files created with any text editor.

used to indicate how often the Testbed's operation should be suspended so

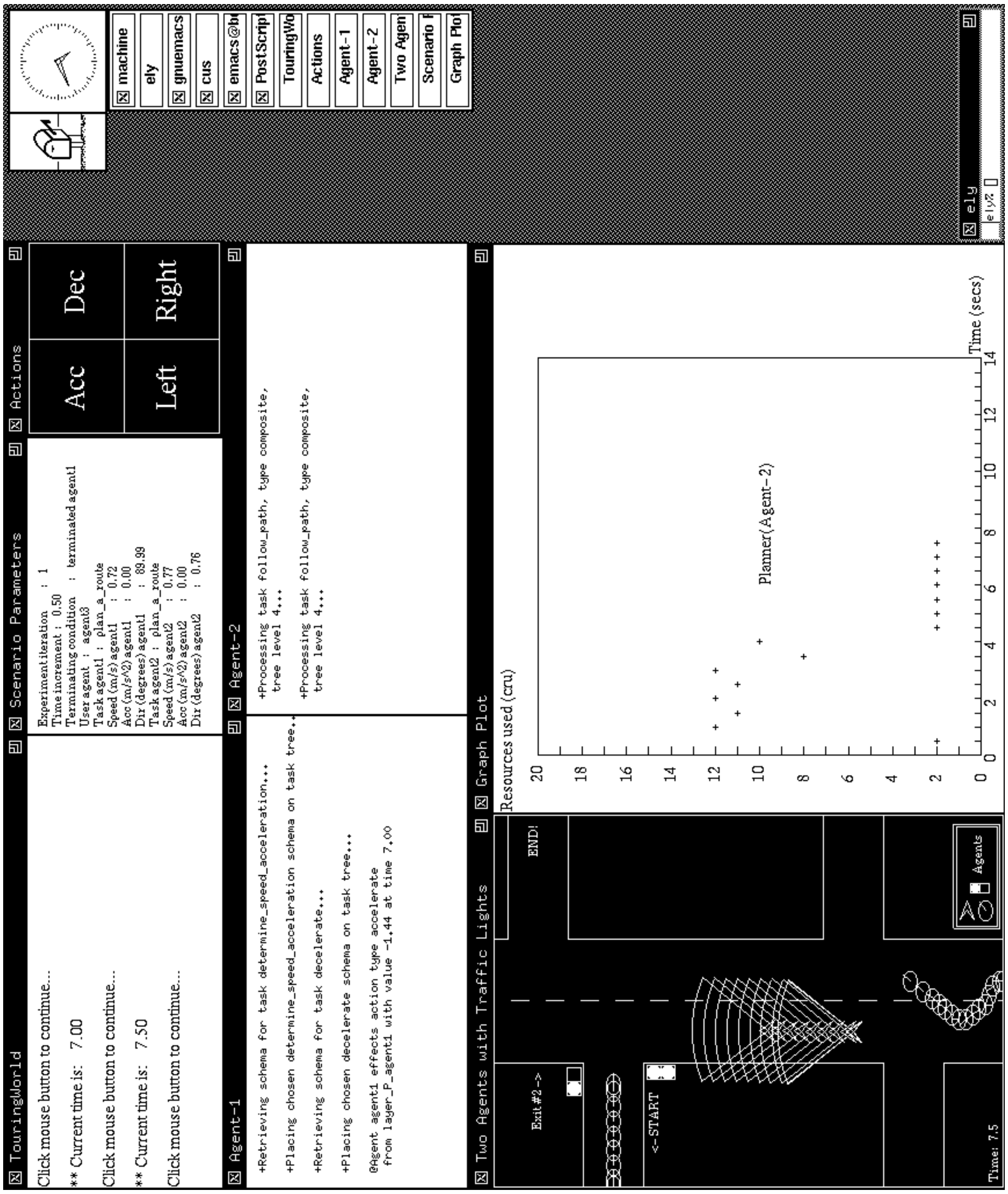

Figure 7.4: Testbed user interface showing various types of input/output windows: textual trace (`TouringWorld`, `Agent-1`, and `Agent-2`), plotted graph (`Graph Plot`), user-driven agent control (`Actions`), scenario parameter display (`Scenario Parameters`), and graphics (`Two Agents with Traffic Lights`).

that, for instance, the user can take stock of what is being displayed on the screen; the Testbed will resume operation once the user clicks the mouse in an appropriate window (titled `TouringWorld` — see Figure 7.4). The **TerminationCriterion** declaration allows the user to specify the condition under which the current scenario iteration should be automatically ended. Termination can be specified to take place upon the first collision occurring, upon a given agent terminating (either by colliding or by arriving at its goal), upon all agents terminating, or upon the passing of a given length of simulated world time.

7.4.1.2 RunWorld

The main function of the *RunWorld* process is to initialise and administer the various entity processes that are present in the scenario. In particular, RunWorld keeps a constant check on the simulated world clock to determine whether there are any agents listed in the initial Scenario Specification which have yet to have had their corresponding entity processes activated and placed on the Testbed's process scheduler queue (see Figure 7.3). An entity process is activated if the starting time in the corresponding entity specification (see Section 7.4.3 below) matches the current simulated world time. Likewise, RunWorld is also charged with monitoring for entity processes which are to be terminated: at the beginning of each simulation cycle, RunWorld will determine if any agents have recently terminated and then remove their corresponding entity processes from the process scheduler queue.

One of the main administrative duties performed by RunWorld includes the setting up of appropriate I/O channels for each entity process. Specifically, RunWorld will be called upon to set up textual output windows for some of these processes (see, for example, windows `Agent-1` and `Agent-2` in Figure 7.4). Additionally, if one of the scenario agents is identified as being a *user-driven-agent* (see Section 7.4.3 below), it will need to set up a suitably defined `Actions` input window (see Figure 7.4, for example) and then regularly poll this window for the user's mouse-clicked agent control commands. Finally, if there is any scenario agent whose *process type* is defined as being `external` (in other words, the entity process is to be executed as a forked UNIX[9] process running a separate Prolog session — see below), RunWorld will be responsible for setting up the necessary collection of semaphored I/O files to "link" such external processes with each of the various (centralised) scenario- and environment-level processes.

Being the process charged with the administration of user-driven agents,

[9] *UNIX* is a Trademark of Bell Laboratories.

RunWorld is also responsible for providing a facility with which the user can automatically record (that is, save to a file) descriptions of all of the actions effected by the user-driven agent during the current Testbed session. Additionally, because each of the saved action descriptions is time-stamped with the precise value that the simulated world clock (see Section 7.4.2) displayed when the action was originally effected, the saved file can, in any future scenarios, be "played back" by the user so that the user-agent executes precisely the same actions that it had in the past. This facility is controlled via the **UserPlayback** parameter whose arguments specify whether the facility is currently being used and if so, whether it is recording current or playing back previous user-agent actions.

Other parameter declarations that fall within the scope of the RunWorld process include: a set of optional **TraceParameter** declarations which instruct RunWorld to output certain items of program trace to appropriate scenario- and entity-level windows (the full list of traceable items is defined under *trace-parameter* in Appendix A, page 188); zero or more **ParmsWindowEntry** declarations which indicate those Testbed *(parameter name, value)* pairs which should be displayed in the `Scenario Parameters` window (see Figure 7.4, for example); and **SaveScript** which indicates whether the program trace information — that which was generated by the presence of particular **TraceParameter** declarations — should also be saved to a file for future reference by the user.

7.4.1.3 HandleStats

The *HandleStats* process provides a facility for collecting and computing statistics about certain aspects of the scenario that the user currently happens to be running. In particular, HandleStats enables the automatic gathering of a number of important statistics regarding the environment (for example, how many agents have collided or how many agents have successfully achieved their tasks) as well as about each individual agent (for example, the utilisation of resources by a particular control layer, the total number of goal conflicts detected or resolved, the total number of reactive rule firings, the agent's average speed of travel).

As illustrated in Figure 7.3, HandleStats is invoked only after the various entity- and environment-level processes have finished running. Which statistics get collected and how they are to be computed is determined by the (optional) presence of one or more **RecordableParameter** declarations in the Scenario Specification file. Each declaration specifies the name of a particular Testbed statistic to be computed (the full list is defined under *recordable-tmw-parameter* in Appendix A, page 187), its scope (which scenario agents the collection of statistics applies to), and its style. The style specification indicates whether the statistic is `incremental` (its value will be re-computed every timeslice) or `non-incremental` (its value will be computed once at the end of each scenario iteration).

In addition, HandleStats also provides a facility for displaying a graphical plot of any collected statistics. In particular, the user can define, through a set of SSL declarations, a number of attributes pertaining to a window-based graphical plot: the X and Y variables (Testbed parameters) to be plotted; whether the graph should be updated every timeslice or at the end of each scenario iteration; the origin, range, and unit increment size for each coordinate axis; as well as textual labels for the two axes and the graph curve. In Figure 7.4, for example, the `Graph Plot` window is being used to show how, throughout the course of a particular scenario iteration, one agent's planner's computational resource usage varies during each timeslice. The full set of declarations for defining a graphical plot are defined under *graph-plot-declarations* in Appendix A, page 184.

7.4.1.4 DrawGraphics

The *DrawGraphics* process is responsible for maintaining an up-to-date graphical image of any entity-level activities which take place during the running of the scenario. The facility is particularly useful for testing and debugging different agent configurations as it provides the user with immediate visual feedback concerning the various actions and state changes which are taking place in the environment.

DrawGraphics associates with each entity type a particular graphical image: object types have associated built-in images (for example, obstacles appear as solid circles, lane markings appear as dashed lines) whereas the shape of each agent — `circle`, `triangle`, or `enterprise` (chevron-shaped) — can be defined via an **AgentGraphicsShape** declaration. At the end of each simulation cycle — that is, after all other Testbed processes have been run — DrawGraphics outputs an appropriate image for each entity present in the scenario. Output is written to a dedicated graphics window (see Figure 7.2, for example) which is set up by the SessionHandler process at the beginning of the session.[10]

The initial set of entities to be displayed is determined through inspection of the various object and agent definitions that appear in the Scenario Specification file. Once the scenario is under way, however, DrawGraphics must make use of the World State fact base produced by the WorldUpdater process

[10] DrawGraphics output is ignored when the Testbed is being run on a line-oriented terminal.

(see below) as this contains the most up-to-date description of the current environmental layout, including agents' new whereabouts and state descriptions. State descriptions indicate, for instance, whether an agent has collided or whether it has somehow left the scenario.

Other Testbed parameters that fall within the scope of the DrawGraphics process include: **GraphicsWindowCoordinates**, which specifies the initial two-dimensional area of the environment to be displayed; **ScrollGraphics**, which specifies the frequency and the (possibly zero) amount by which the graphics window should be vertically scrolled; as well as a set of zero or more **DrawForwardSensingArc** and **DrawRearSensingArc** declarations that are used to specify which named agents, if any, should have their sensing arcs visibly displayed in the graphics window (see, for example, the chevron-shaped agent which appears in the graphics window of Figure 7.4).

7.4.2 Environment-level Processes

7.4.2.1 WorldUpdater

WorldUpdater, the only environment-level process, can be considered the main simulator process since, included among its responsibilities, are the creation and maintenance of the *World State* — a fact base containing up-to-date descriptions of each TouringWorld entity's changing state and physical configuration. The World State fact base is created at the beginning of the Testbed session by taking copies of certain pieces of information from the definitions of entities appearing in the Scenario Specification file. In particular, World State fact base records are created for each scenario entity and contain such information as the entity's name, physical dimensions, and initial Cartesian location. In addition, if the entity is a mobile agent, a record is also made of its initial speed, acceleration, and orientation, plus any communicated information that might initially be associated with it (for example, whether the agent is signalling or honking its horn).

Once the scenario gets under way, the WorldUpdater's main function becomes that of plausibly simulating each of the scenario agent's different actions. Implemented as entity-level processes (see below), agents send messages to the WorldUpdater process whenever they have opted to effect some physical or communicative action. Once every timeslice, after the entire collection of entity processes has been suspended from executing, WorldUpdater proceeds to deal with the various agent action messages in one single batch. Simulation of these actions is achieved by making appropriate updates to the World State fact base. In particular, by taking into account agents' most recent configurations (as recorded in the current World State) plus any actions which they might have submitted during the present timeslice, WorldUpdater will compute their respective quadratic motion trajectories (agents move with uniform acceleration according to the standard equations of motion $s = v_0t + \frac{1}{2}at^2$ and $v = v_0 + at$) and then use these space-time trajectories to determine and record agents' new configurations for consideration during the next timeslice.

Although environmental concurrency is readily mimicked through the round-robin scheduling of processes and through the implementation of agents as processes that are regularly suspended at the end of every timeslice (see below), there is also a need to ensure that any actions taken by the scenario agents are physically plausible. In the TouringWorld Testbed this is achieved by taking into account each of the agent-agent and agent-object *collisions* that would occur if each agent were to move along the particular motion trajectory that WorldUpdater had recently calculated for it. Collisions are detected by performing pair-wise intersections of all of the agent space-time trajectories and then selecting the "valid" intersection or *collision points*. A collision point is valid if it occurs during the simulator's present timeslice; if it occurs after the present timeslice, it will be ignored until some future timeslice. Also, if several such collision points involving the same agent are found, then the one that occurs earliest in time is the one selected: an agent's trajectory may intersect the trajectories of several agents at a number of different points, but, since agents always come to a halt after any collision, only the earliest among these should be considered valid. Eventually, once all of the trajectory intersections have been performed and appropriate collision participants determined, WorldUpdater will make all necessary updates to the World State fact base. In addition, it will inform the RunWorld process about any terminated agents so that their corresponding entity processes can be removed from the process scheduler queue.

Parameters pertaining to the WorldUpdater process include: **InitialWorldTime**, the starting value for the simulated world clock; **WorldTimeIncrement**, the discrete time quanta through which the world clock should be advanced at the end of each process scheduler cycle; **FogFactor**, a real number between 0.0 and 1.0 which is altered by the fog agent (as described in Section 7.3) and which is used by WorldUpdater to increase or reduce the sensing range of any agent that happens to fall within the fog agent's zone of application (see Figure 7.2); **RainFactor**, a similar parameter, this one altered by the rain agent and used by WorldUpdater to increase or reduce the effective braking distance of any agent which happens to be braking within the rain agent's zone of application. A number of other parameters also exist which enable the user to specify the granularity or precision with which certain Testbed calculations should be performed. In particular, **DistanceTruncation**, **SpeedTruncation**, **AccelerationTruncation**, and **AngleTruncation** declarations are used to specify the precision with which the WorldUpdater process should compute new values for agents' Cartesian location coordinates, speeds, accelerations, and orientations, respectively. Thus, if required, the 2-dimensional space of the TouringWorld could be made "chessboard-like" if the **DistanceTruncation** factor was chosen such that it rounded all distance calculations to a whole number of units of distance and if the **AngleTruncation** factor was chosen so that it forced each angle calculation to one of the four normal angles 0°, 90°, 180°, or 270°. This would then facilitate comparisons with other grid-based agent testbeds (for example, DVMT [LC83] and MICE [DM89]).

7.4.3 Entity-level Processes

Entities are the various objects and agents that "inhabit" the TouringWorld environment. Each object is specified via an *object-definition* (see Appendix A, page 189) which comprises the following parameter declarations: **ObjectName**, **ObjectType** (for example, `obstacle`, `wall`, `kerb`), **ObjectLocation** (Cartesian coordinates in the TouringWorld environment), **EntitySize** (radius size if an obstacle, linear length if a wall, kerb, or lane marking), and a number of optional **ObjectAttributes** (for example, `vertical` or `horizontal` if the object is a wall, `dashed` or `solid` if it is a lane marking). Because objects remain inactive throughout the duration of a scenario, they are not, unlike agents, implemented as separate system processes. Rather, their physical presence is simply noted in the World State fact base for subsequent use by the various entity and WorldUpdater processes (for example, to perform simulated sensing or to handle collisions).

Entity processes, then, implement TouringWorld agents. As noted in Section 7.3, these agents can be mobile (standard or user-driven TouringMachines) or environmental (traffic lights, fog, and rain).[11] Each agent type has a slightly different set of defining parameters. The simplest among these are the environment agents, which, although task-achieving, are devoid of such components as sensors, focussing mechanisms, and reactive or modelling control layers. The parameters used for specifying environment agents are defined in Appendix A, page 189, and include **EnvironmentAgentName**, **AgentProcessType** (`internal` or `external` — see below), **EnvironmentAgentVector** (stating its start time, end time, and Cartesian location), optional **EnvironmentAgentAttributes** (for example, size and orientation if it is a traffic light), **EntitySize** (radius size if the agent is fog or rain), plus various planning layer declarations which are common to both environment and mobile

[11] At present, TouringWorld scenarios may contain at most one fog agent and one rain agent.

agent types (see below).

The parameters used for describing standard mobile agents are divided into groups corresponding to the various "physical" components that go into making TouringMachines. Thus, in addition to several top-level entity parameters such as **AgentName**, **AgentProcessType** (see below), **AgentVector**, and **EntitySize**, a host of other parameters (see Appendix A, pages 189–191) are provided for describing TouringMachines, including:

- various physical, kinematic, and mediatory control parameters: **MaxSpeed**, **MaxAcceleration**, **MinAcceleration**, **MaxTurningRate** (maximum rate at which the agent's steering wheel can be turned), **CommsDeviceStatus** (specifying whether or not the agent's indicators, brake lights, etc. are operational), and a set of **CensorControlRule** and **SuppressorControlRule** declarations (control rules implementing the agent's mediatory control framework);

- sensor definitions: **SensingAlgorithm** (one of eight different algorithms simulating some form of restricted — see Figure 7.5 — or unrestricted "sensing" on the World State fact base), **ForwardSensingRange**, **ForwardSensingArc**, **RearSensingRange**, **RearSensingArc**, and **SensingRate** (how frequently the World State fact base is sensed);

- focus of attention definitions (common to layers $\mathcal{P}$ and $\mathcal{M}$): **FocussingRules** (initial set of focussing rules used), **FocussingResources**, **FocussingEntityCost** (computational resource cost to focus on any single entity), and **FocussingFlagCost** (extra resource cost to handle dynamic focussing requests from layers $\mathcal{P}$ or $\mathcal{M}$);

- layer $\mathcal{R}$ definitions: a set of initial reactive control rules, together with value declarations for the different parameters referred to in these rules (for example, **KerbAvoidanceAngle**, **ObstacleAvoidanceAngle**, plus various others described in Section 4.3);

- layer $\mathcal{P}$ definitions: **PlannerAlgorithm** (a pointer to the code implementing the agent's layer $\mathcal{P}$ functions),[12] **PlannerResources**, **PlannerTask** (a specification of the task to be carried out), **SchemaRetrievalCost** (resource cost to retrieve a single schema from the Schema Library), **SchemaPlacingCost** (resource cost to place a retrieved schema

[12]Code hooks have been provided at various points in the Scenario Specification in order to make it easier in the future to experiment with different implementations of certain TouringMachine components (for example, the Planner) or to make it simpler to add new instances of some existing components (for example, sensing algorithms, focussing predicates, reactive rules).

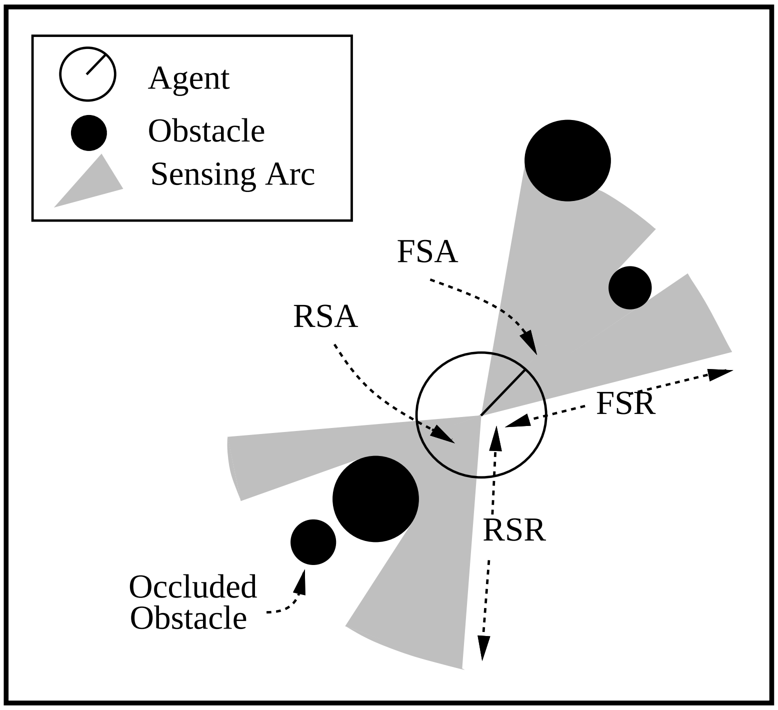

Figure 7.5: Illustration of an agent's sensory field (shaded grey area) when its sensors are programmed with the algorithm `all-but-occluded`. Specifically, this means that the agent will be able to sense entities both to its rear and to its front, but that it will not be able to detect entities that are occluded. (Other Testbed sensing algorithms are listed in Appendix A, page 194.) Also illustrated are some of the agent's other sensor parameters: **ForwardSensingRange** (FSR), **ForwardSensingArc** (FSA), **RearSensingRange** (RSR), and **RearSensingArc** (RSA).

in the Planner's plan structure), **HasSchemas** (the set of schemas contained in the agent's Schema Library), and a Topological World Map made up of a number of **TouringWorldPath** and **TouringWorldJunction** declarations);

- layer $\mathcal{M}$ definitions: **ModellerAlgorithm** (a pointer to the code implementing the agent's layer $\mathcal{M}$ functions), **ModellerResources**, **ModelRetrievalCost** (resource cost to retrieve a model from the Model Library), **ModellerFlagCost** (resource cost to process conflict detection flags from the agent's effectors or layer $\mathcal{P}$), **ConflictDetectionHorizon** (temporal range, expressed in number of whole timeslices, over which to look for potential goal conflicts), **ConflictResolutionDepth** (number of levels of counterfactual reasoning agent should undertake when looking for such goal conflicts), **ModellingRate** (frequency with which the agent's models should be updated with fresh sensory input), **ModelDiscardAfterTime** (length of time to maintain a model for an entity which is no longer registered by the agent's sensors), **ModelLocationBounds** (+/− value bounds placed on the expected future location of some other agent), **ModelSpeedBounds** (ditto for an agent's expected speed), **ModelAccelerationBounds** (ditto for an agent's expected rate of acceleration, **ModelOrientationBounds** (ditto for an agent's expected orientation), **HasModelTemplates** (the set of model templates contained in the agent's Model Library), a set of **TheoristKBaseEntry** declarations (causal rules the agent can use when ascribing intentions to other world entities), a set of **SpaceTimeProjection** declarations (functions the agent can use when projecting other entities' movements through space-time), and a set of **ConflictResolutionRule** declarations (rules the agent can use when identifying and resolving different intra- and inter-agent goal conflicts).

A TouringMachine can also be user-driven. As mentioned above, the user controls such an agent by clicking the mouse in a particular Testbed window titled `Actions` (see Figure 7.6). The `Actions` window is divided into four quadrants, each acting as an "active button" which, when selected with the mouse, will send a specific physical action command to the user-driven agent's Action Buffer. At present, the available actions include accelerate, decelerate, turn left (counterclockwise), and turn right (clockwise); also associated with these actions are a series of parameters which are used in defining their specific scalar values: **UserAccelerationIncrement**, **UserDecelerationIncrement**, **UserTurnLeftIncrement**, and **UserTurnRightIncrement**, respectively. User-driven agents (see Appendix A, page 189) are completely controlled by the user and therefore lack such capabilities as focussing of attention, planning, and modelling. As described above, user-driven agents can alternatively be controlled by "playing back" a file of time-stamped action descriptions which the user has opted to record during an earlier Testbed session.

As mentioned above in the description of the RunWorld process, there are two types of entity processes: *internal* and *external* (these are defined via the parameter **AgentProcessType**). An internal entity process is implemented as a Prolog program module which is part of the same main testbed process (a UNIX process running a Prolog session) as the various scenario- and environment-level processes. An external process, on the other hand, is a separate UNIX process, forked from the main testbed process, which runs independently but which communicates with the main process (the one that is running the scenario-, environment-, and internal entity-level processes) via a set of semaphored I/O files. Via these files, each external entity both receives its necessary input — a complete copy of the current World State fact base so that it can determine the whereabouts and physical extent of other

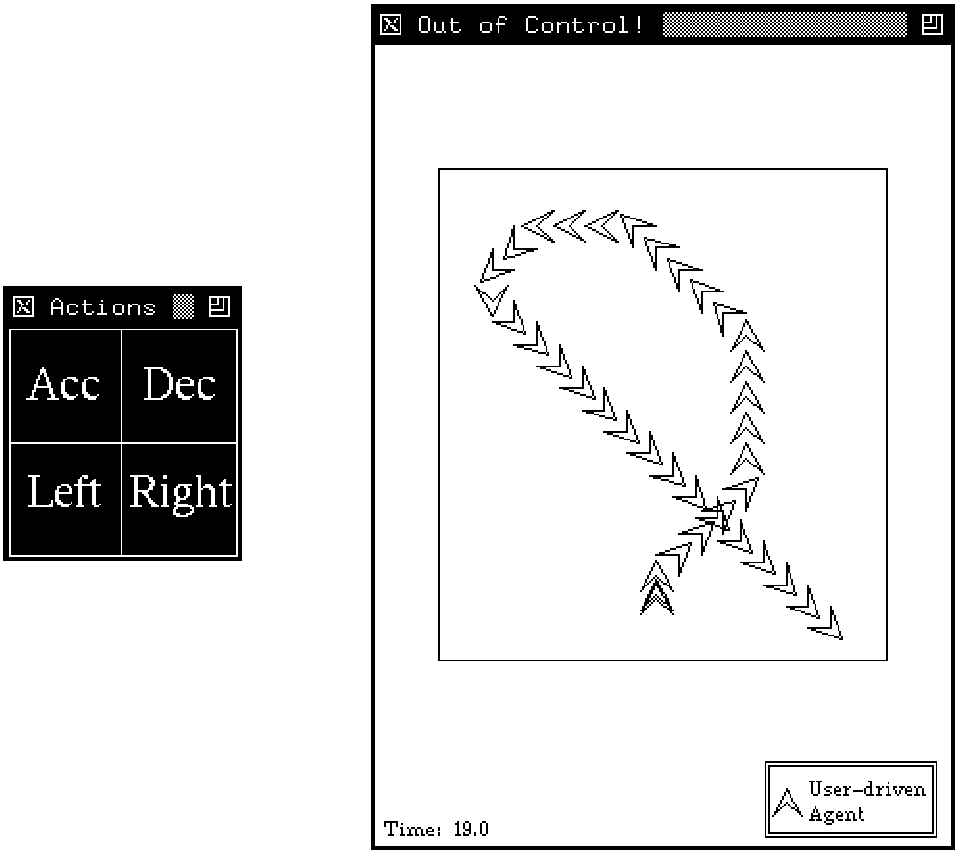

Figure 7.6: The Testbed user can control a specified TouringMachine's acceleration and orientation by clicking the mouse in appropriate sections of the `Actions` window. In addition, the precise sequence of control commands can be automatically recorded and then played back verbatim in subsequent scenarios.

scenario entities — and sends its corresponding output, including effected actions (sent to WorldUpdater), tracing information (sent to RunWorld), plus any relevant statistical information (sent to HandleStats). The advantage of using external processes to implement TouringWorld entities is that they can help speed up operations in the Testbed by increasing throughput: external processes are sent their input before any internal processes are considered and have their output collected only after each one of the internal processes has been executed. On the other hand, because of the overhead associated with file I/O between external entity processes and the main testbed process, entity processes are best defined to be internal if the number of entities in the scenario is small.

Since the entity processes representing TouringMachines are executed on a sequential machine running a sequential language (Prolog), a couple of design features are included to ensure that the Testbed provides a plausible simulation of the TouringMachine's' concurrent activities. The first feature is aimed at providing a plausible simulation of the concurrent flow of messages between control layers that regularly takes place within a TouringMachine. To achieve this effect — in other words, to ensure that any messages sent during one timeslice are not processed by the receiver until at least the next timeslice — messages are always time-stamped before being sent to another layer. These time-stamps are then used by receiving layers to test whether the input message currently being considered was sent prior to the present timeslice or not. If it was, then the message is removed from the message input queue and duly processed; if it was not, the message is left on the queue so that it will be re-considered in some future timeslice. The second design feature aims to ensure that each entity process gets a fair share of CPU ownership during each timeslice: that is, during each and every cycle of the process scheduler queue. This is the same as guaranteeing an upper-bound on each entity's per-timeslice inter-operation latency which, as described above, is achieved through the explicit use of computational resource units for monitoring the costs of each of the operations performed by TouringMachines. In particular, an entity process representing a TouringMachine will be forced to relinquish CPU control (that is, become suspended until the next timeslice or scheduler cycle) either when it has used up its limited per-timeslice resources (these are applied to each of the process' sensing, focussing, planning, and modelling operations) or when the entity has submitted some physical or communicative action to the WorldUpdater process.

Unlike scenario- and environment-level processes which are *statically* defined before runtime, entity-level processes are created (and terminated) *dynamically* — once the scenario has started. Specifically, processes representing the various entities defined in the Scenario Specification file are created whenever an entity's start time (one of the arguments to its **AgentVector** or **EnvironmentAgentVector** parameter declaration) coincides with the time displayed by the simulated world clock. As described above, the RunWorld process is responsible for both initialising the entity process (in particular, adding it to the process scheduler queue) and for terminating the entity process (removing it from the queue) whenever the corresponding entity collides or successfully achieves whatever task it was assigned.

One of the principles applied throughout the design and implementation of the Testbed was to promote and maintain a reasonably clean interface between the TouringWorld entities and their environment so that, if necessary, the Testbed could be used for running and testing alternative implementations of agents without too much effort on the part of the programmer. In fact, providing such a clean I/O interface was needed anyway in order to be able to support the external entity process facility described above. All the same, while in terms of its range and richness of both simulator and user-level features the TouringWorld Testbed can be compared quite favourably with other DAI or integrated agent testbeds — for example, DVMT [LC83], Phoenix [CGHH89], MICE [DM89], SeaWorld [VB90], and Tileworld [PR90] — it should be clear that it is not intended as a fully general agent programming environment. Rather, the TouringWorld Testbed should be regarded, primarily, as a vehicle for evaluating TouringMachines. Describing several such evaluations will be the focus of the next chapter.

8

Evaluating TouringMachines

Ordinarily a computer user would construct a problem, feed it in, and wait for the machine to calculate its solution — one problem, one solution ... [The] chaos researchers ... needed more. They needed to do what Lorenz had done, to create miniature universes and observe their evolution. Then they could change this feature or that and observe the changed paths that would result. They were armed with the new conviction, after all, that tiny changes in certain features could lead to remarkable changes in overall behavior.

James Gleick, *Chaos*

8.1 Purpose of Experimentation

There has, in recent years, been a proliferation of intelligent agent architectures, each one offering a different perspective on the problem of how to integrate a variety of intelligent control functions in a single autonomous, computational system. The ways in which one agent architecture might be considered better than another, Drummond [DK90] argues, are not always very clear. As a result, there has been a growing realisation that many of the characteristics, both positive and negative, of any particular agent architecture, only become evident when experimental evaluation is performed. Indeed, it could be argued — as it has been already by a number of researchers in the agent design and related AI subfields [DK90, LD90, Coh91] — that to progress as a science, we must develop more rigorous experimental methods.

Unfortunately, AI, according to Cohen [Coh91, page 35], is "unlike experimental sciences that provide editorial guidance and university courses in experiment design and analysis." Indeed, at present, there is no common language or frame of reference for describing and assessing different agent architecture designs and their performances. Much like the AI subfield of planning, agent architecture design is, as an experimental science, entering the initial stages of its evolution. As such, Langley [LD90, page 113] would argue, researchers should be satisfied at present with identifying "qualitative regularities that show one method as better than another under certain conditions, or that show one environmental factor as more devastating [on system performance] than another." Only in the later stages of this evolution should researchers focus their experiments to determine the quantitative laws that can actually *predict* agent architecture performance.

The primary aim behind experimenting with the TouringMachine agent architecture, then, will be to test and substantiate the main hypothesis or thesis stated at the beginning of this dissertation: namely, that is it both desirable and feasible to combine non-deliberative and suitably designed and integrated deliberative control functions in a single — hybrid — architecture in order to obtain effective, robust, and flexible behaviours from rational, autonomous, resource-bounded agents which are to carry out their tasks in complex domains. Another aim behind such experimentation will be to investigate a secondary hypothesis which is that establishing an appropriate balance between reasoning (deliberative control) and acting (non-deliberative control) depends heavily on characteristics of the task environments in which the agents are intended to operate. To investigate this latter claim, it is argued, one must seek to obtain an improved understanding of TouringMachine *behavioural ecology*.

8.1.1 Behavioural Ecology of TouringMachines

One useful approach toward understanding the reasons for the behaviours exhibited by the TouringMachine agent design — and, more specifically, for identifying the conditions under which one configuration of the architecture performs better than another — is to vary the environment in which it operates. The simplest approach to this issue, Langley [LD90] argues, involves designing a set of benchmark problems, of which some, for the purposes of scientific comparison (that is, for the purposes of enabling independent variation of different task environment attributes), should involve artificial domains. The TouringWorld environment is one such domain (other examples, as mentioned in previous chapters, include the Phoenix environment [CGHH89], the Tileworld [PR90], and MICE [DM90]).

The power of the TouringWorld Testbed domain, and of artificial domains in general, arises from the insights it can provide toward the improved understanding of agent — in this case, TouringMachine — behavioural ecology: in other words, the understanding of the functional relationships that exist between the designs of agents (their internal structures and processes), their behaviours (the tasks they solve and the ways in which they solve these tasks), and the environments in which they are ultimately intended to operate [CGHH89].

The characterisation of TouringMachines as a study of agent behavioural ecology exemplifies a research methodology which emphasises complete, autonomous agents and complex, dynamic task environments. Within this methodological context, the focus of the present evaluation has been centred on two particular research tasks. Cohen et al. [CGHH89] refer to these as *environmental analysis*, in other words, understanding what characteristics of the environment most significantly constrain agent design; and the *design task*, in other words, understanding which agent design or configuration produces the desired behaviours under the expected range of environmental conditions.

These two tasks, in fact, are the first two stages of a more complete research methodology which Cohen [Coh91] refers to as the *MAD* methodology, for *modelling*, *analysis*, and *design*.[1] This methodology aims to justify system design (and re-design) decisions with the use of predictive models of a system's behaviours and of the environmental factors that affect these system behaviours. Like IRMA agents in the Tileworld domain [PR90], TouringMachine agents can be viewed as having been developed via an incremental version of MAD, in which the (causal) model of TouringMachine behaviour is developed incrementally, at the same time as the agent design. In other words, the agent design (or some part of its design) is implemented as early as possible, in order to provide empirical data (or feedback) which flesh out the model, which then become the basis for subsequent redesign [Coh91]. The implications of adopting such a design method, as well as the roles played in this method by the environmental and behavioural analyses referred to above, will be considered in more detail in Section 8.4.

8.1.2 Some Methodological Issues

The present evaluation of TouringMachines will be realised through a series of interesting task scenarios involving zero or more agents and/or zero or more obstacles or traffic lights. The scenarios have been selected with the

[1]The remaining design activities — *predicting* how the system (agent) will behave in particular situations, *explaining* why the agent behaves as it does, and *generalising* agent designs to different classes of systems, environments, and behaviours — are beyond the scope of this dissertation. See Cohen [Coh91, pages 29–32] for details.

aim of evaluating some of the different capabilities and behaviours which TouringMachines will require if they are to complete their tasks in a competent and effective manner — for example, reacting to unexpected events, effecting of goal-directed actions, reflective and predictive goal monitoring, spatio-temporal reasoning, plan repair, coping with limited computational and informational resources, as well as dealing with real-time environmental change. The scenarios can be considered interesting because they succinctly exercise agents' abilities to carry out time-constrained tasks in complex — partially-structured, dynamic, real-time, multi-agent — environments. Although the chosen scenarios are simplified to deal only with mentally and structurally homogeneous agents possessing noiseless sensors, perfect actuators, and approximately similar non-shared relocation tasks (such simplifying assumptions are discussed in full in Section 7.3), these still present a number of non-trivial challenges to TouringMachine agents.

It is not the aim of the present evaluation to show that the TouringMachine architecture is in any sense "optimal". As argued in Section 3.5, optimal rational behaviour will in general be impossible if the agent is resource-bounded, has several goals, and is to operate in a real-time multi-agent environment in which events are able to take place at several levels of space-time granularity. As such, one should more realistically expect a TouringMachine to behave satisficingly, but at times — for example, when under extreme real-time pressure — to fail to satisfy every one of its outstanding goals. What is really of interest here is understanding how the different configurations of agents and the different environmental characteristics to which such configurations are subjected affect, positively or negatively, the ability of agents to satisfy their goals.

It is also not the aim of the present evaluation to show that TouringMachines are "better" than other integrated agent architectures at performing their various tasks. Rarely is it the case that the actual and/or intended task domains of different agent architectures are described in sufficient detail so as to permit direct comparisons of agent performance. The lack, at present, of any common benchmark tasks or of any universally agreed upon criteria for assessing agent performance — previous evaluations have relied either on a single performance criterion (for example, the total point score earned for filling holes in specific single-agent Tileworld environments [PR90, KG91]), or on a small number of performance criteria which can only be interpreted with respect to the particular architecture being measured (for example, the total number of behaviours communicated between agents in selected MICE environments [DM90]) — combine to make detailed *quantitative* comparisons with other architectures extremely difficult if not altogether impossible.

Due to the relatively large number of parameters which the TouringWorld Testbed provides for specifying different agent configurations, performance evaluation criteria (for example, task completion time, resource utilisation), and agent task and environmental characteristics (see Appendix A), the present evaluation will necessarily be partial, the main focus being placed on studying selected *qualitative* aspects of TouringMachine behavioural ecology — namely, some of the effects on agent behaviour which, in a given task environment, can occur through varying individual agent configuration parameters; and the effects on agent behaviour which, for a given agent configuration, can occur through varying certain aspects of the agent's environment. Like with the Tileworld experiments described by Pollack and Ringuette [PR90, page 187], a number of TouringWorld "knobs" (for example, world clock timeslice size, total per-timeslice resources available to each agent, agent size, agent speed and acceleration/deceleration rate limits, agent sensing algorithm, initial attention focussing heuristics, reactive rule thresholds, plan schema and model template library entries) have been set to provide "baseline" environments which are dynamic, somewhat unpredictable, and moderately paced. In such environments, a competent (suitably configured) agent *should* be able to complete all of its goals, more or less according to schedule; however, under certain environmental conditions and/or agent parametrizations — a number of which will be analysed below — this will not always be the case. In order to simplify the analysis of agents' behaviours in *multi-agent* settings, TouringMachine configurations — both mental and physical — should be presumed identical unless otherwise stated.

8.2 Single-agent Scenarios

8.2.1 Reflective Goal Monitoring: sensitivity to model discrepancies

In monitoring its own state, and in particular, in determining whether the model it maintains of its own current physical configuration (its location, speed, orientation, etc.) is as it should be — that is, satisfies the expectations which were computed when it last projected its own self model in space-time — a TouringMachine makes use of various tolerance bounds to decide whether any model discrepancies in fact exist. Identification of a self model discrepancy typically requires further investigation to determine its possible cause. Often, this reasoning process will result in the agent's layer $\mathcal{M}$ flexibly resolving the conflict by submitting an appropriate "corrective" action command to the agent's effectors. For example, a discrepancy between the agent's current

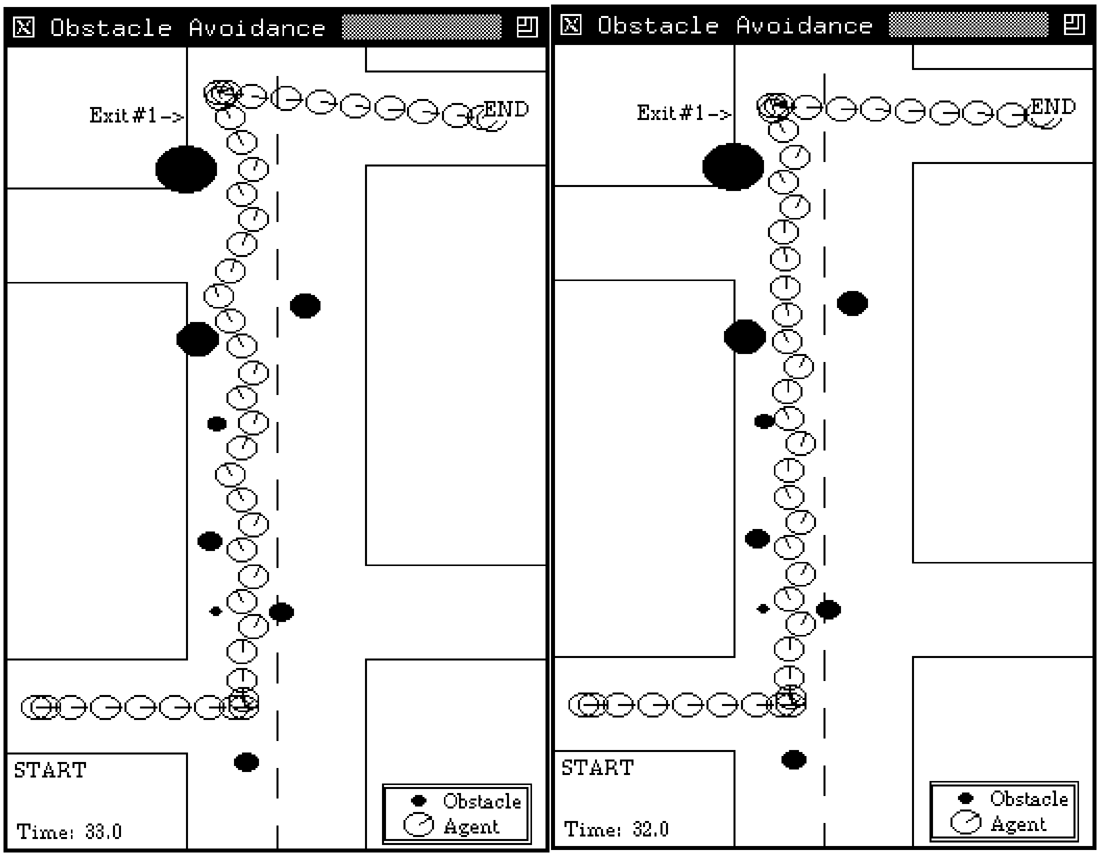

Figure 8.1: Sensitivity to self model orientation discrepancies can be varied through the layer $\mathcal{M}$ parameter **ModelOrientationBounds**.

and expected speeds might be indicative of an unforeseen stoppage or slow-down (perhaps due to the presence of some other agent or obstacle) which, potentially, might threaten any of the agent's time-constrained goals unless some suitable and contextually appropriate action is taken.

In Figure 8.1 we can see the effect on the agent's behaviour — that is, on its ability to carry out its relocation task `reach-destination` in an effective and robust manner — of modifying the value of one of its self model tolerance bounds, **ModelOrientationBounds**, which is used to constrain the allowable deviations between the agent's current heading and its erstwhile desired heading. With fairly wide bounds (for example, **ModelOrientationBounds** = +/- 40°), the agent fails to notice any changes in its current orientation — in this scenario caused by the agent's layer $\mathcal{R}$ reacting to obstacles, kerbs, and lane markings — and so does not take any corrective re-orientation actions (Figure 8.1, left-hand frame). As a result, and compared to the situation when the agent is configured with **ModelOrientationBounds** = +/-0°, and therefore extremely sensitive to any orientation discrepancies (Figure 8.1, right-hand frame), the agent with wide orientation bounds covers more distance and so arrives at the target destination `END` slightly later (specifically, after a total of 33 seconds instead of 32).

This in itself, of course, does not suggest that agents should always be configured with tight orientation bounds. Other factors might need to be considered here such as the importance of the agent's deadline (for example, it could be that a small delay is tolerable), the number of resources that are being spent each time a corrective action is taken, and, perhaps not so obviously, the physical structure of the environment in which the agent is operating (for example, when navigating along narrow paths, very little wandering is possible regardless of which orientation tolerance bounds are in use; however, if these paths were very wide it might become more important to prevent or minimise wandering).

8.2.2 Predicting Possible Goal Conflicts: efficiency versus reliability

To construct, at time T_0, expectations of its own physical configuration for some future time T_F, a TouringMachine projects its own self model (together with any models which it currently maintains for any other world entities) up to some point in time, T_{CDH} ($T_{CDH} \geq T_F$), to determine if any goal conflicts are likely to occur. Predicted conflicts, for instance, might include a *physical* collision with another world entity (the agent's goal in conflict in this case would be `avoid-collisions`) or a *virtual* collision which might occur, for example, if the agent were to run through a red traffic light (the goal in conflict here would be `obey-regulations`). The length of time $|T_{CDH} - T_0|$ over which the agent will make such predictions is controlled by the layer $\mathcal{M}$ parameter **ConflictDetectionHorizon**. (In fact, the value of this parameter is expressed as the integer number of processing cycles or timeslices equivalent in length to $|T_{CDH} - T_0|$.)

In Figure 8.2 we can see the effect on an agent's behaviour — that is, on its ability to respond flexibly to a change in colour by a traffic light — of modifying the value of its **ConflictDetectionHorizon** parameter.[2] The two upper frames of Figure 8.2 show two snapshots — the left-hand one at time T = 8.5 seconds, the right-hand one at T = 10.5 seconds — when the agent is configured with **ConflictDetectionHorizon** = 1. The two lower frames show

[2]Since traffic lights are agents with plans (albeit very simple ones), this example must really be seen as a multi-agent scenario and so should therefore be discussed in Section 8.3. However, since the emphasis of the present analysis will be on the behavioural characteristics of the sole TouringMachine, the scenario can more usefully be viewed as being single-agent.

two snapshots — the left-hand one at time $T = 7.5$ seconds, the right-hand one at $T = 10.5$ seconds — when the agent is configured with **ConflictDetection-Horizon** = 3. In each case, the traffic light — the rectangular entity, one half of which contains a black circle — changes from red (circle in upper half of rectangle) to green (circle in lower half) at time $T = 8.0$.

If the agent's **ConflictDetectionHorizon** parameter is set to 1, it detects a *possible* virtual conflict at time $T = 8.5$ — exactly one timeslice before $T = 9.0$ which is when the agent will start to drive past the physical location of the light; that is, when the conflict, if one is deemed to exist, will take place. Now, since the light, as mentioned above, changes to green at $T = 8.0$, no conflict will in fact be predicted to occur, so the agent will simply proceed through the light and on toward its ultimate destination (Figure 8.2, upper right-hand frame). If, on the other hand, the agent is configured with **ConflictDetectionHorizon** = 3, the agent will detect a *definite* virtual conflict at time $T =$ 7.5, since, at that point in time, the traffic light will still be red (it will not turn green for another 0.5 seconds). To resolve the impending `obey-regulations` conflict, the agent's layer $\mathcal{M}$ will propose to the agent's layer $\mathcal{P}$ to adopt the new intention `stop-at-light`. The interesting thing in this situation, however, is that by the time the agent actually comes to a halt at $T = 10.5$ (Figure 8.2, lower right-hand frame), the traffic light will have already changed back to green (it did so exactly one timeslice *after* the agent had committed itself to stopping), making any actions taken by the agent to resolve the initial conflict completely worthless. Thus, with this latter more predictive configuration (**ConflictDetectionHorizon** = 3), the agent clearly behaves less effectively than its former less predictive self (with **ConflictDetectionHorizon** = 1).

This scenario is illustrative of a more general problem confronted by predictive agents — Shoham and McDermott [SM90] refer to it as the *extended prediction problem* — namely, that of choosing the length of time intervals in the future to which predictions should refer. This is a problem because it involves a tradeoff between efficiency and reliability: conservative predictions (for example, when **ConflictDetectionHorizon** = 1) refer to relatively short intervals of time which, by definition, will make it hard to reason about lengthy time periods; on the other hand, more ambitious predictions (for example, when **ConflictDetectionHorizon** = 3) can be unreliable because they cover larger intervals of time during which the world is more likely to change (potentially invalidating any predictions that might have been made in the meantime). In the scenario of Figure 8.2, the ideal value for the agent's **ConflictDetectionHorizon** parameter would appear to be 1 since, as described, this does not cause the agent to take any unnecessary actions.

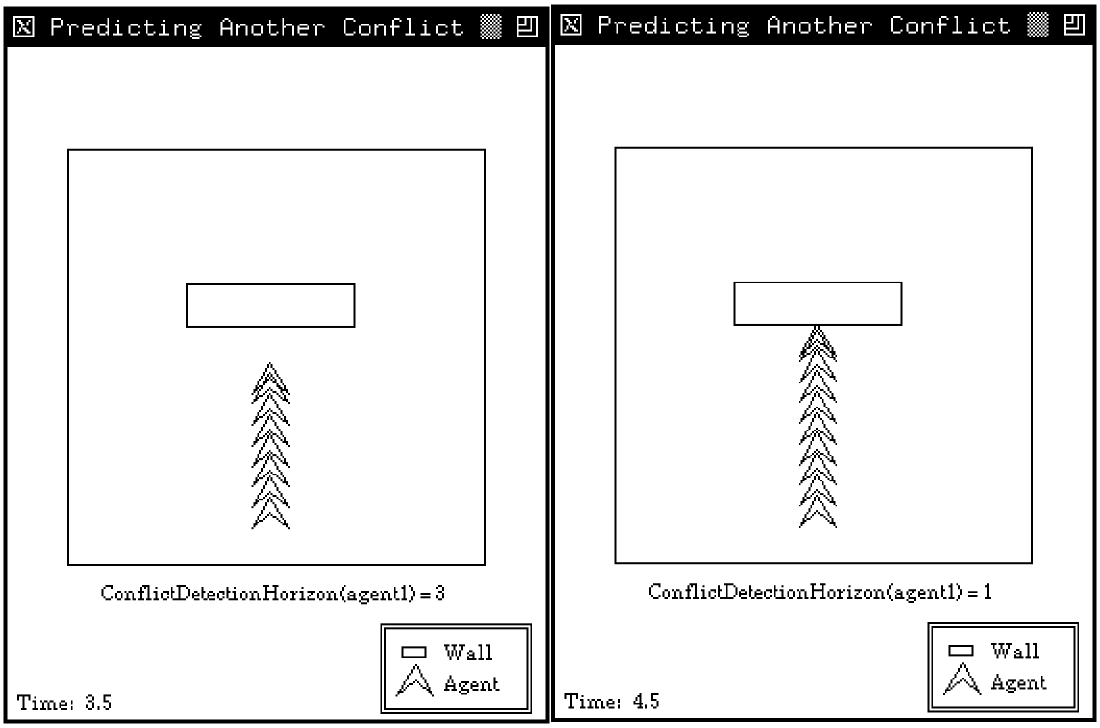

Figure 8.3: Possible effects of varying an agent's **ConflictDetectionHorizon** parameter — part 2.

In general, however, this value may not be the most effective for all environmental contexts. Figure 8.3, for example, depicts the same agent of Figure 8.2, this time heading straight for a wall which is blocking its path. In this case, it is the configuration with **ConflictDetectionHorizon** = 3 (left-hand frame of Figure 8.3) rather than that with **ConflictDetectionHorizon** = 1 (right-hand frame of Figure 8.3), that is best able to deal with the agent's impending `avoid-collisions` goal conflict; this will occur at around time T = 4.5 if no action is taken. Although the agent requires only one timeslice (0.5 seconds) to come to a halt, detecting the impending conflict at time T = 4.0, which is what happens when the agent is configured with **ConflictDetection-Horizon** = 1 — as opposed to at time T = 3.0, which is what happens when **ConflictDetectionHorizon** = 3 — is simply too late to avoid the collision. This would seem to indicate, therefore, that the effectiveness of a particular agent configuration will depend, to some extent at least, on characteristics of the environment in which the agent is operating.

8.2.3 Emergent Behaviour: choosing the "right" configuration

In the absence of any knowledge regarding the whereabouts of other agents or static obstacles, a TouringMachine's layer $\mathcal{P}$ will construct a shortest length route to its target destination (or to the next junction to turn at if the agent's route comprises more than one path). If the agent encounters any obstructions en route (for example, if it encounters an obstacle), it will, provided it has the appropriate layer $\mathcal{R}$ reactive rule, avoid hitting such obstructions by effecting suitably robust changes in orientation (the magnitude of such changes is set by the agent parameter **ObstacleAvoidanceAngle**). In response to these directional changes, however, and provided the agent's layer $\mathcal{M}$ is suitably sensitive to orientation discrepancies arising in its own self model (this, as described above, is set by the parameter **ModelOrientationBounds**), the agent will flexibly counteract with subsequent (re-)orientation actions in order to adjust its heading back toward the direction of the target destination. Whether an action command originating from the agent's layer $\mathcal{R}$ is chosen over an action command originating from its layer $\mathcal{M}$, ultimately depends on the programming of the agent's mediatory suppressive control rules. In the TouringWorld domain (and therefore in all of the scenarios presented in this chapter) layer $\mathcal{R}$ reactions to nearby obstacles will always be favoured over any corrective actions proposed by layer $\mathcal{M}$; in other words, short-term robustness will be favoured over more longer term flexibility.

In Figure 8.4 we can observe the progress of an agent toward a target destination, marked by X, which happens to lie on the far side of a fairly big obstacle. Two configurations are tested here: the first — see upper left-hand frame of Figure 8.4 — with the physical capability parameter **ForwardSensingArc** set to 30° (an explanation of this parameter is given in Section 7.4.3), the second — see upper right-hand frame of Figure 8.4 — with **ForwardSensingArc** = 25°. In the former case, the agent, through a combination of robust reactions to the obstacle and corrective re-orientations toward the target X, ends up describing a more or less circular path, effectively following the contour of the obstacle. In the latter case, with the narrower forward sensing arc, the agent ends up following a more or less rectilinear path, eventually colliding with the obstacle after only a few timeslices of motion.

This scenario demonstrates that particular combinations of "primitive" behaviours — for example, obstacle avoidance (effected by layer $\mathcal{R}$) and corrective goal re-orientation (effected by layer $\mathcal{M}$) — can be used to exploit certain structural characteristics or regularities in the environment and, as a consequence, to produce desirable and effective *emergent* behaviours such as the contour- or wall-following one observed in the upper left-hand frame of Figure 8.4. However, as seen from the upper right-hand frame of Figure 8.4, it would appear that other aspects of the agent's physical configuration (for example, its particular sensing capabilities) can affect how or whether such emergent behaviours ultimately arise. This is more easily explained with the use of the lower two frames of Figure 8.4 which show the same two agent configurations described

above only now displayed with their corresponding forward sensing arcs. With the wider sensing arc (lower left-hand frame of Figure 8.4), the agent ends up reacting away from the obstacle more often than it re-orients itself toward its goal, for the simple reason that it senses the obstacle more often than not. And since, as mentioned above, reactions are always favoured to goal re-orientations, the agent will end up reacting more often and so, in this environmental context, avoid colliding with the obstacle. With the narrower sensing arc, however (see lower right-hand frame of Figure 8.4), the agent is less likely to see the obstacle after a reaction away from it (the obstacle falls outside the area described by its sensing arc) and so is more likely to re-orient itself more often back toward its target goal. In fact, in the case where the agent is configured with **ForwardSensingArc** = 25°, the agent ends up re-orienting itself toward it goal exactly as often as it reacts away from it, and so, as illustrated, will follow a more or less rectilinear path in the direction of the obstacle.

8.3 Multi-agent scenarios

8.3.1 Counterfactual Reasoning: why modelling other agents' intentions can be useful

In constructing and projecting models of other world entities, a TouringMachine must constrain its modelling activities along a number of dimensions. Implemented as user-definable parameters, these layer $\mathcal{M}$ constraints have been described in earlier chapters, and some, for example **ModelOrientationBounds** and **ConflictDetectionHorizon**, have also been addressed in this chapter. One layer $\mathcal{M}$ parameter which has not been mentioned in this chapter is **ConflictResolutionDepth** — the parameter which fixes the number of levels of counterfactual reasoning the agent should undertake when projecting entities' models to discover possible future goal conflicts. In general, when constructing model projections at counterfactual reasoning level N, an agent will take into account any conflicts *plus any actions resulting from the anticipated resolutions to these conflicts* which it had previously detected at level $N-1$. Values of **ConflictResolutionDepth** which are greater than

1, then, give agents the flexibility to take into account — up to some fixed number of nested levels of modelling — any agent's responses to any other agent's responses to any predicted conflict.

In the scenario of Figure 8.5, two TouringMachine agents can be seen following independent routes to one destination or another. The interesting

agent to focus on here — the one whose configuration is to be varied — is `agent1` (the round one). The upper left-hand frame of Figure 8.5 simply shows the state of the world at time T = 15.5 seconds. Throughout the scenario, each agent continually updates and projects the models they hold of each other, checking to see if any conflicts might be "lurking" in the future. At T = 17.5 (upper right-hand frame of Figure 8.5), `agent1` detects one such conflict: an `obey-regulations` conflict which will occur at T = 22.0 between `agent2` (chevron-shaped) and the traffic light (currently red). Now, assuming `agent1` is just far enough away from the traffic light so that it does not, within its parametrized conflict detection horizon, see any conflict between itself and the traffic light, then, if `agent1` is configured with **ConflictResolutionDepth** = 1, it will predict the impending conflict between `agent2` and the traffic light, as well as the likely event of `agent2` altering its intention to `stop-at-light` so that it will come to a halt at or around T = 22.0. If, on the other hand, `agent1` is configured with **ConflictResolutionDepth** = 2, not only will it predict the same conflict between `agent2` and the traffic light and the resolution to be realised by this entity, but it will also, upon hypothesising about the world state *after* this conflict resolution is realised, predict another impending conflict, this second one involving itself and the soon to be stationary `agent2`.

The observable effects of this parameter difference are quite remarkable. When `agent1` is configured with **ConflictResolutionDepth** = 1, it will not detect this second conflict — the one between itself and `agent2` — until one timeslice later; that is, at time T = 18.0 instead of at T = 17.5. Due to the proximity of the two agents, the relatively high speed of `agent1`, and the inevitable delay associated with any change in intention or momentum, this 0.5 second delay proves to be sufficiently large to make `agent1` realise too late that `agent2` is going to stop; an inevitable rear-end collision therefore occurs at T = 22.0 (Figure 8.5, lower left-hand frame).[3] Configured with **ConflictResolutionDepth** = 2 (Figure 8.5, lower right-hand frame), `agent1` ends up having enough time — an extra 0.5 seconds — to adopt and realise the appropriate intention `stop-behind-agent`, thereby avoiding the collision

[3]In fact, this collision need not be "inevitable": in this scenario both `agent1` and `agent2` have been configured with fairly insensitive (not very robust) layer $\mathcal{R}$ reactions, primarily to emphasise the different behaviours that *could* result from different parametrizations of agents' modelling capabilities.

that would otherwise have occurred.

Having the flexibility to reason about the interactions between other world entities (for example, between `agent2` and the traffic light) and to take into account the likely future intentions of these entities (for example, `stop-at-light`) can enable TouringMachines like `agent1` to make timely and effective predictions about the changes that are taking place or that are likely to take place in the world. In general, however, knowing how deeply agents should model one another is not so clear: since the number of layer $\mathcal{M}$ resources required to model world entities is proportional to both the *number* of entities modelled and the (counterfactual reasoning) *depth* to which they are modelled, agents will ultimately have to strike a balance between breadth of coverage (more entities modelled, little detail) and depth of coverage (less entities, more detail). This issue is re-visited later in Section 9.2.

8.3.2 Monitoring the Environment: sensitivity versus efficiency

In monitoring the state of another world entity, and in particular, in determining whether the model it maintains of an entity's current physical configuration (its location, speed, orientation, etc.) is as it should be — that is, satisfies the expectations which were computed when it last projected the entity's model in space-time — a TouringMachine makes use of various tolerance bounds to decide whether any discrepancies in fact exist. As with any discrepancies detected in the agent's self model (see Section 8.2.1 above), identification of a discrepancy in the model of another entity typically requires further investigation to determine its cause. Often this reasoning process results in having to re-explain the entity's current behaviour by ascribing it a new intention. For example, a discrepancy between the modelled entity's current and expected speeds might be indicative of the entity's change of intention from, say, `drive-along-path` to `stop-at-junction`.

In Figure 8.6 (upper two frames) we can see, at two different time points, $T = 12.5$ seconds and $T = 15.5$ seconds, several agents in pursuit of their respective goals: `agent1` (round), `agent2` (chevron-shaped), and `agent3` (triangular, top-most). Furthermore, we can see the effect on `agent1`'s behaviour — that is, on its ability to carry out its homeostatic goal `avoid-collisions` — of modifying the value of **ModelSpeedBounds**, the parameter which, when modelling another entity, is used to constrain the "allowable" deviations between this entity's currently observed speed and the speed it was predicted to have had when the entity was last observed. In this scenario, `agent1` has to contend with the numerous and unexpected speed changes effected by `agent2`, a user-driven agent. With fairly tights bounds (for example **ModelSpeedBounds** = +/-0.5 ms^{-1}), `agent1` detects any speed discrepancies in `agent2` which are greater than or equal to 0.5 ms^{-1}. Among such discrepancies detected by `agent1` are those which result from the `agent2`'s deceleration just prior to its coming to a halt at a junction at time T = 20.0 (Figure 8.6, lower left-hand frame). As a result, and compared to the situation when `agent1` is configured with **ModelSpeedBounds** = +/-2.0 ms^{-1}, and therefore, in this particular scenario, unable to detect or respond fast enough to `agent2`'s actions at T = 20.0 (Figure 8.6, right-hand frame), the configuration with tighter speed bounds is more robust, more able to detect "important" events (for example, the agent in front coming to a halt) and also more able to carry out timely and effective intention changes (for example, from `drive-along-path` to `stop-behind-agent`).

This in itself, of course, does not suggest that agents should always be configured with tight speed bounds. Sensitivity or robustness to environmental change can come at a price in terms of increased resource consumption: each time an agent detects a model discrepancy it is forced by design to try to explain the discrepancy through a (relatively expensive) process of abductive intention ascription. Often, however, small changes in the physical configuration of a modelled entity need not be the result of the entity having changed intentions. In the scenario of Figure 8.6, for example, `agent2`'s speed changes are due entirely to actions effected by the user. Ignorant of this, however, `agent1` configured with **ModelSpeedBounds** = +/-0.5 ms^{-1} will continually attempt to re-explain `agent2`'s changing behaviour — despite the fact that this reasoning process will always, except in the case when `agent2` stops at the junction, return the same explanation of `drive-along-path`. It is also important to remember that a TouringMachine may only monitor the state of its own layer $\mathcal{M}$ goals when there are exactly zero discrepancies to attend to in it entire current model set. A less environmentally sensitive agent, therefore, might well end up with more opportunities to monitor its own progress and so, potentially, achieve its goals more effectively.

8.4 Discussion

A number of issues arise directly from the above evaluation of the TouringMachine agent architecture:

- The balance between goal-orientedness (effectiveness) and reactivity (robustness) in an agent can be affected by a number of factors, including the length and depth (level of detail) of the predictions the agent makes (see Section 8.3.1); the size of the model discrepancy tolerance bounds it is configured with (see Section 8.3.2); and also certain aspects of some of its physical capabilities (for example, the scope of its sensing apparatus — see Section 8.2.3). Other factors would include certain environmental characteristics (for example, the rate of change of events or degree of clutter), the sensitivity — or threshold bounds — of the agent's reactive rules, and the proportion of total resources made available for constructing plans or building and maintaining mental models of other agents.

- Predicting future world states through the flexible modelling of agents' mental states, can, in certain situations, prove useful for promoting effective coordination between agents (see, for example, Section 8.3.1). This especially appears to be the case when the particular entity being modelled possesses complex intentions which subsequently cause the entity to effect a series of complex actions in space-time (for example, decelerating over a period of time before coming to a temporary halt at a red traffic light). Without some consideration of entities' likely intentions — and of the causal relationship these intentions have with their assumed goals — predictions of possible future events (for example, impending collisions) will inevitably become shallower and thus prove less useful in promoting effective behaviour.

- There is a trade off between the reliability and the efficiency of the predictions an agent can make about the future. Knowing precisely what to consider when making predictions (which entities to model, what aspects of these entities to model, how deeply to model these aspects) or how far into the future such predictions should be made, would appear to depend, certainly to some extent, on selected aspects of the agent's environment (compare the two scenarios of Section 8.2.2).

- Distinguishing between contextually relevant and irrelevant events, and, more generally, achieving the right balance between robustness to unexpected events (for example, coping with the sudden appearance of another entity or obstacle) and flexibility to environmental change (for example, establishing that an observed agent is not behaving as predicted or that an effected action has not been carried out as intended) would appear, in the TouringWorld , to depend on a number of factors; these include, among others, the physical structure of the environment, the (temporal) criticality of the agent's task (see Section 8.2.1, for example), and the availability of adequate computational resources for flexibly reasoning about different world events (see Section 8.3.2). Identifying the optimal level of "sensitivity" to environmental change has also been recognised as a critical issue in Sanborn and Hendler's Traffic World system [SH88] and, through the use plan-monitoring envelopes, in the Phoenix agent project [CGHH89].

- New and possibly effective behavioural patterns (for example, wall-following) can emerge when differing primitive behaviours (for example, obstacle avoidance and corrective goal re-orientation) are allowed to operate concurrently (see Section 8.2.3, for example). Their precise manifestation, and more importantly, their ultimate usefulness vis-à-vis the successful completion of an agent's goals, would also appear to depend on (structural) characteristics of the environment, as well as certain physical capabilities within the agent itself (for example, it sensing properties).

- Under certain conditions, a TouringMachine can fail to accomplish one or more of its goals. Failure may be outright (when resulting, for example, from a collision) or partial (for example, when measured in terms of the agent's task effectiveness or timeliness). This, as seen above, might be due to a number of factors, including under-sensitivity to environmental change (see Section 8.3.2), short-term or limited predictiveness (see Section 8.2.2, Figure 8.3 and Section 8.3.1), excessive predictiveness (see Section 8.2.2, Figure 8.2), and also by possessing inadequate physical capabilities (see Section 8.4). It is also important to note that, no matter how robust or finely tuned a TouringMachine's reactive control layer might be, it will always take a certain amount of time to respond to particular threats. As Brooks [Bro86] suggests, no agent is invincible and the presence of a sufficiently fast-moving entity or a very cluttered environment may well result in the agent colliding.

Now, apart from matters arising directly from the evaluation presented above, a number of experiential and implementational issues which bear on the applicability and appropriateness of the TouringMachine architecture also merit addressing at this point. As mentioned earlier in Section 8.1.1, the first stage in designing the TouringMachine architecture involved an analysis of the intended TouringMachine task environment: that is, a characterisation of those aspects of the intended environment which would most significantly constrain the TouringMachine agent design. As we shall now see, the main purpose of this analysis was to differentiate between, and therefore establish, what Cohen [Coh91] terms the system's *fixed* and *reviewable* design decisions.

Fixed design decisions are those which "will not be reviewed anytime soon" [Coh91, page 31]. In the TouringMachine architecture, these design decisions were established upon close examination of the intended TouringWorld domain.[4] For instance, the decision to divide control among multiple, independent concurrent layers was influenced by the fact that TouringMachines would have to deal flexibly and robustly with any number of simultaneous events, each occurring at a potentially different level of space-time granularity. Such differences in event granularity, together with the need for carrying out both long-term, deadline-constrained tasks, as well as short-term reactions to unexpected events, ultimately played a part in the decision to combine both deliberative and non-deliberative control functions into a single hybrid architecture. In turn, the need to ensure that any such (non-real-time) deliberative functions be "suitable" for use in a real-time domain such as the TouringWorld — in other words, that they be efficient and effective on the one hand but flexible and robust on the other — suggested that such deliberative functions be: *(i)* latency-bounded in order to provide guaranteed system responsiveness (this in turn demands fairly strict control over internal computational resource use); *(ii)* that they operate incrementally (in other words, that they be capable of suspending operation and state after regular — and arbitrarily small — periods of processing time); and *(iii)* that they serve merely as *resources* for action rather than as strict *recipes* for overall agent control. This last requirement would also become the main motivating force behind the decision to employ a context-sensitive mediatory control policy for establishing control layer priorities. Other design decisions worth mentioning here include the incorporation of functions for reasoning about — or modelling — other agents' actions and mental states and for identifying and flexibly resolving conflicts within and between agents (this is necessary because the TouringWorld domain is populated by multiple intentional agents with limited computational and informational resources); and mechanisms for constantly sensing and monitoring the external world (which are needed since the TouringWorld domain is both dynamic and unpredictable).

Identification and isolation of the second type of design decisions, reviewable decisions, are those which, as their name suggests, can be "reviewed after they are implemented" [Coh91, page 31]. The purpose of differentiating fixed and reviewable design decisions was to enable the basic (fixed) design to be implemented and run as early as possible, and to provide an empirical environment in which to develop iteratively this basic agent model, to test hypotheses about how the model should behave, and then to review, subsequently, particular design decisions in the light of observed performance. Also, by providing — in addition to the TouringMachine agent architecture — a highly parametrized and controllable testbed environment like the Tour-

[4]Much of the justification for the basic TouringMachine design has already been presented in Chapter 2 (Section 2.5) and Chapters 3 through 6. What is described here can be regarded simply as a summary of some of the main design decisions that were made.

ingWorld, a very effective and productive framework in which to carry out such design activities was thus established. Examples of reviewable TouringMachine design decisions — established with empirical feedback gained through use of the TouringWorld Testbed — include the particular set of reactive rules initially made available to the agent, the contents of its various domain-specific libraries (plan schemas, model templates, conflict-resolution methods, and space-time projection functions), the initial set of heuristics used to program the agent's focus of attention mechanisms, and the precise set of inhibitory and suppressive control rules which are used to mediate the actions of the agent's three control layers.

To summarise then, while more experimentation would certainly be required before any strong claims could be made concerning the general applicability and appropriateness of the TouringMachine design, there would appear from the above evaluation to be sufficient, albeit tentative, evidence to be reasonably confident about the validity of the claims of this dissertation. In particular, the evaluation of TouringMachines appears to support the claim that it is both desirable and feasible to combine non-deliberative and suitable designed and integrated deliberative control function in a single, hybrid, autonomous agent architecture. As shown, the resulting architecture, when suitably configured, is capable of effective, robust, and flexible behaviours in a reasonably wide range of complex single- and multi-agent task scenarios. As also seen above, the behavioural repertoire of TouringMachines is wide and varied, including behaviours which are reactive, goal-oriented, reflective, and also predictive. Furthermore, the evaluation suggests that establishing an appropriate balance between reasoning and acting — that is, between appropriate degrees of deliberative and non-deliberative control — would appear to depend on characteristics of the task environment in which the particular TouringMachine is operating. More generally, and in line with the experiences of both Maes [Mae90] and Pollack [PR90], there is evidence to suggest that environmental factors invariably play an important role in determining which agent configuration or parametrization is the most appropriate for any given situational context. Finally, one cannot underestimate the importance of deploying — from the earliest stages of design — concrete measures for carrying out extensive experimentation. In this respect, the TouringWorld Testbed domain has proved a viable and useful system for evaluating agent performance.

As well as lending weight to the main claim concerning the desirability and feasibility of using a hybrid agent design approach, the present evaluation should also provide useful insights into some important aspects of the behavioural ecology of mobile autonomous agents. In addition, this evaluation should help to lay the foundations for future experimentation, not only with the TouringMachine architecture, but also, and perhaps more importantly, with the architectures of other agent researchers. The recent, and one hopes growing, interest and activity in establishing common benchmark tasks for agents (see Drummond and Kaelbling [DK90], for example) and in defining formal taxonomies of agent environments (see Wilson [Wil90]), provide the reassuring belief that this will one day be possible.

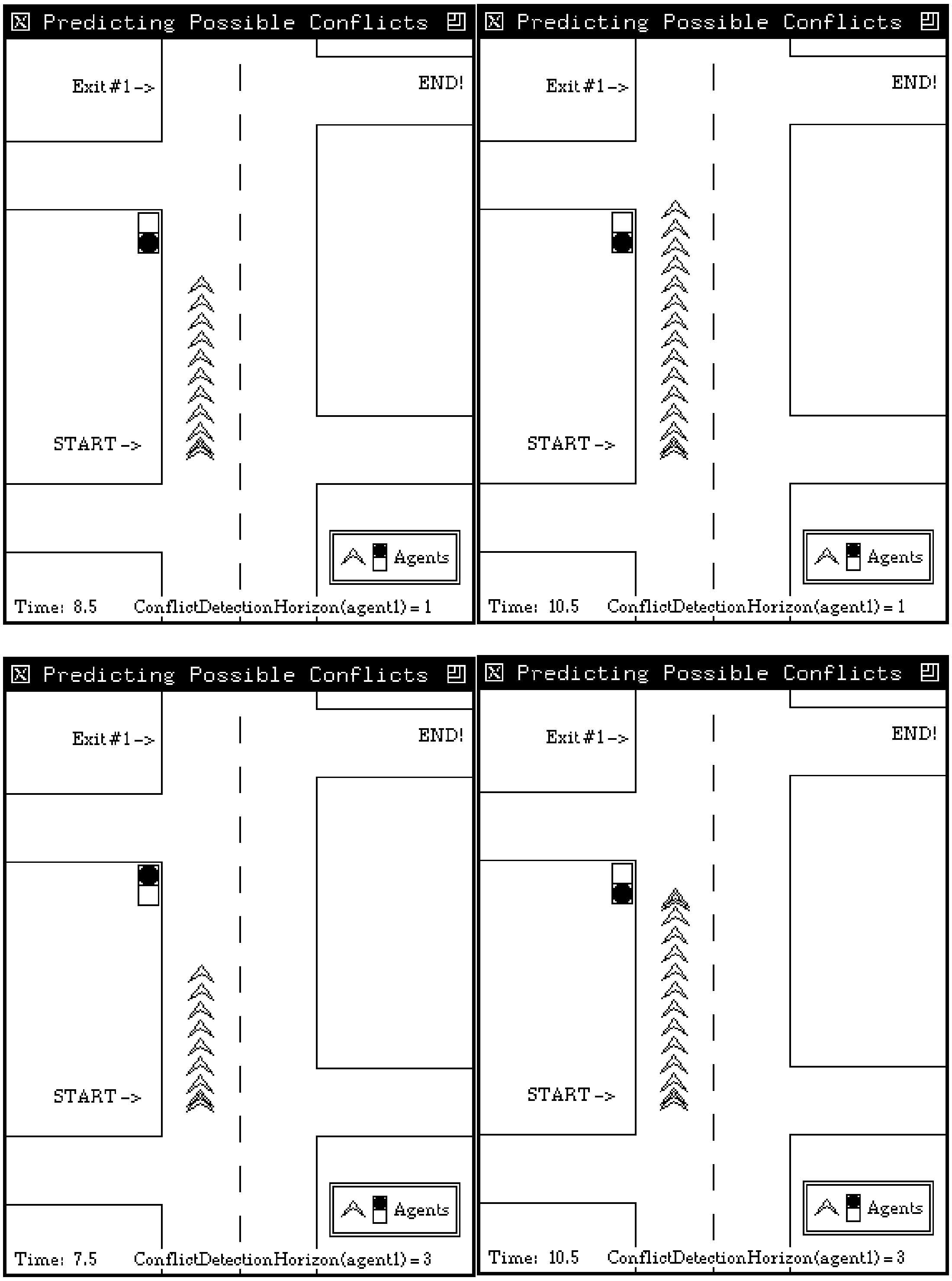

Figure 8.2: Possible effects of varying an agent's **ConflictDetectionHorizon** parameter – part 1.

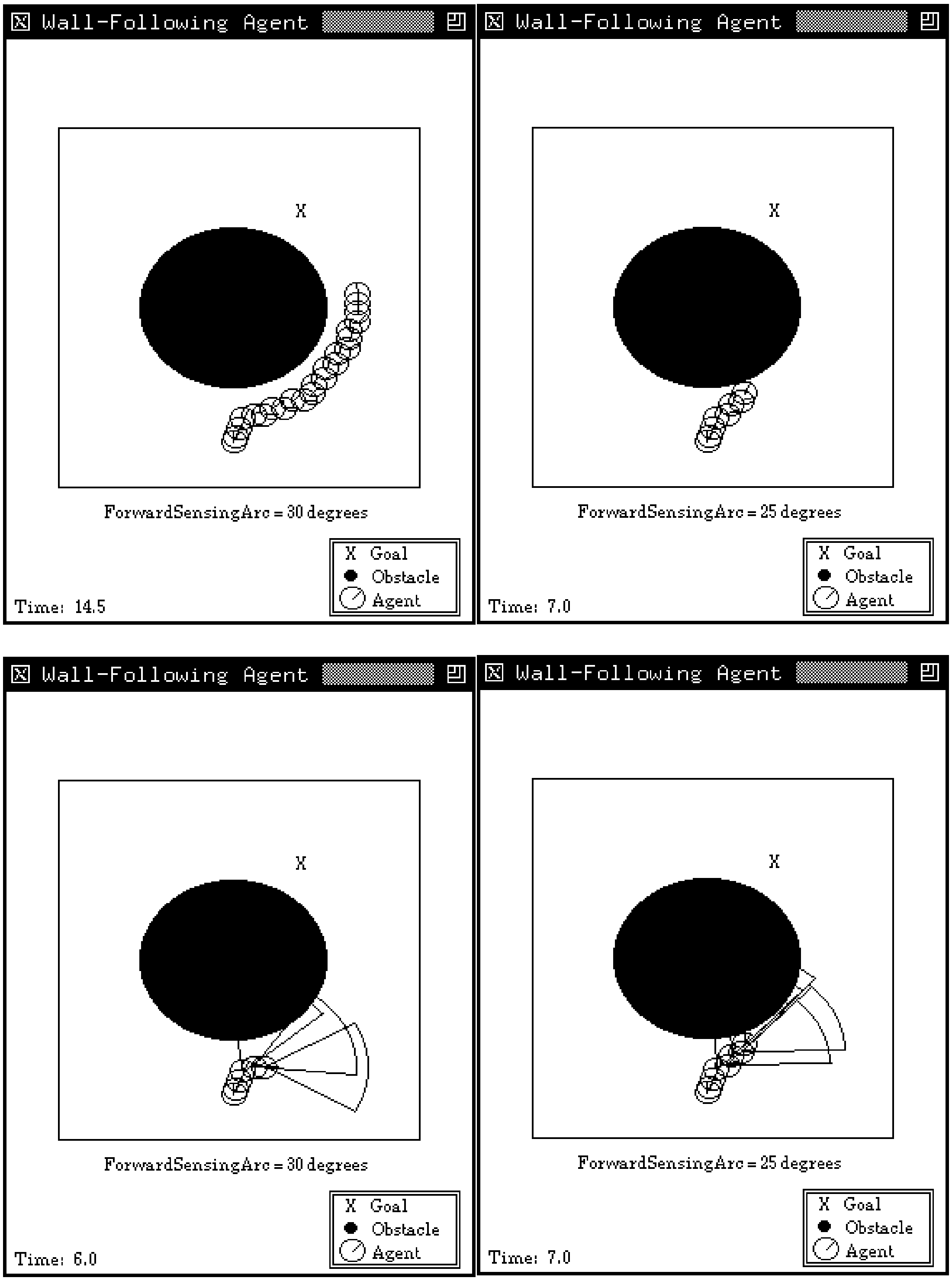

Figure 8.4: Wall-following behaviour can emerge from an appropriate combination of more primitive behaviours (obstacle avoidance and orientation discrepancy correcting), provided the agent has other suitable physical capabilities (wide width of sensory field).

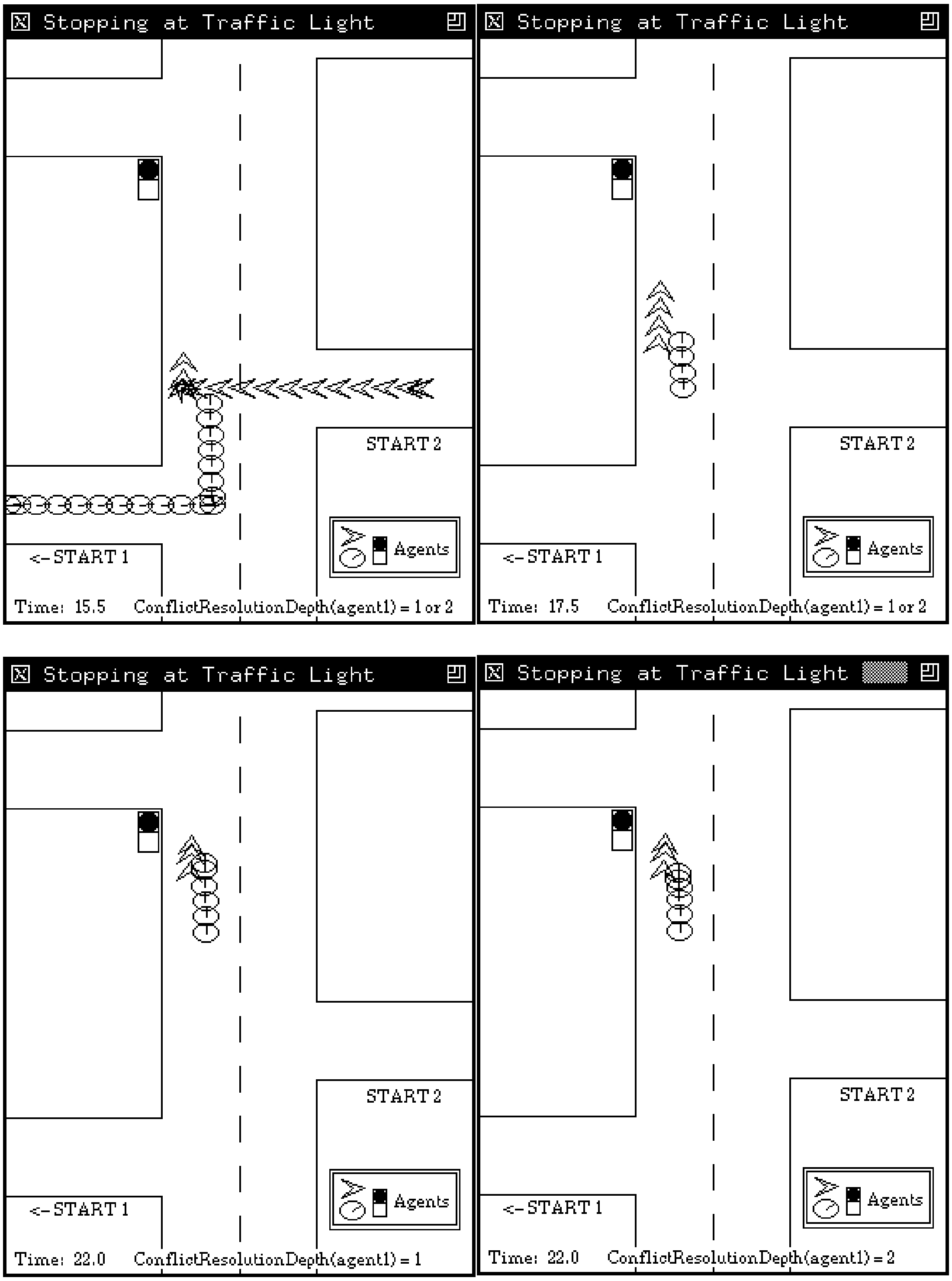

Figure 8.5: Altering the value of an agent's **ConflictResolutionDepth** parameter can affect the timeliness and effectiveness of any predictions it might make.

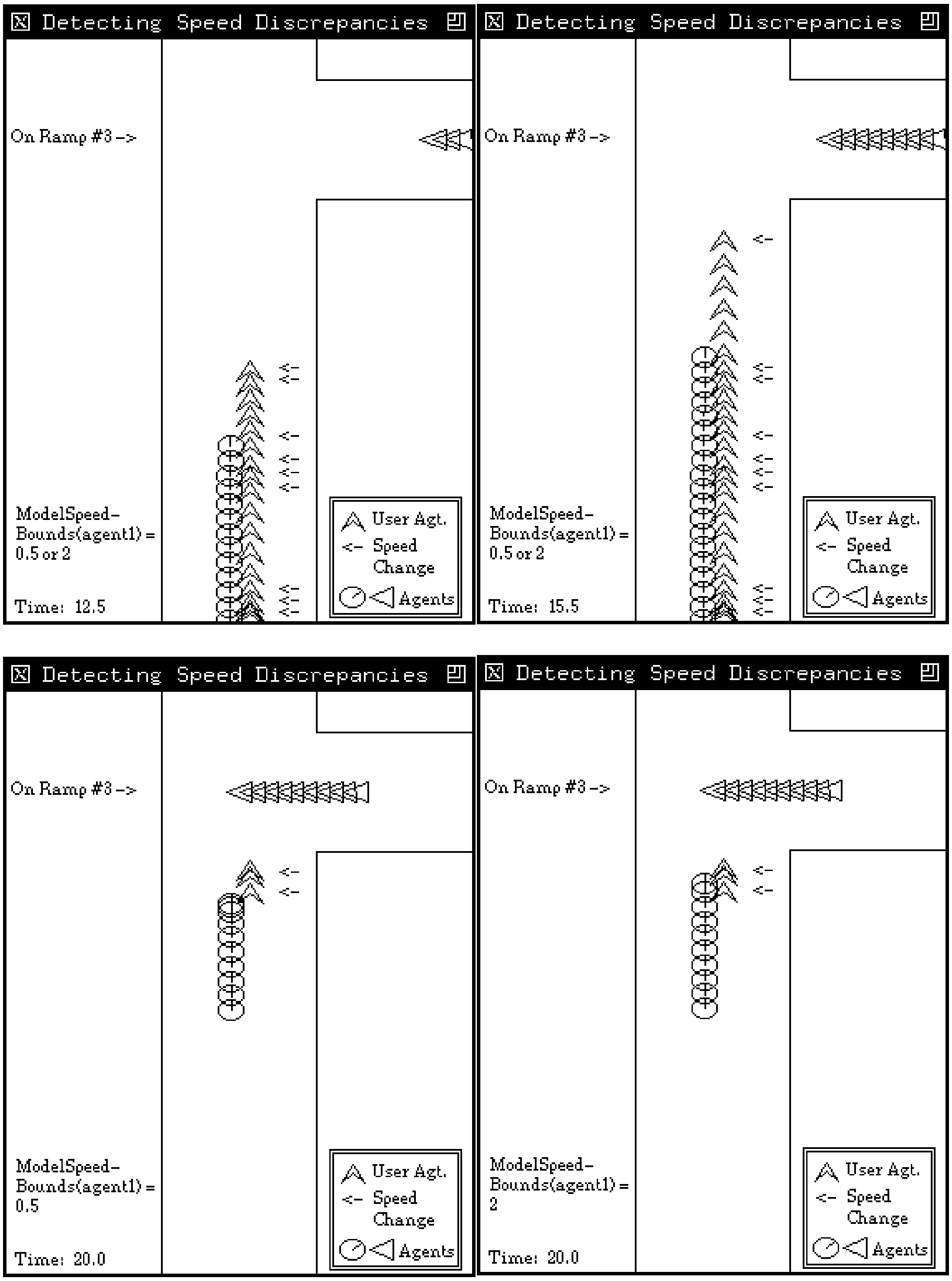

Figure 8.6: Varying the value of an agent's **ModelSpeedBounds** parameter can affect the agent's level of sensitivity to environmental change.

9

Summary and Conclusions

If a machine is expected to be infallible, it cannot also be intelligent.

Alan Turing

9.1 Summary

The research presented in this dissertation is aimed at the design and implementation of an AI software architecture suitable for controlling and coordinating the actions of a rational, resource-bounded autonomous agent embedded in a complex world. The research involved three complementary efforts:

- Understanding the functional and behavioural requirements of intelligent, rational, autonomous, mobile agents for a particular class of dynamic, partially-structured, real-time, multi-agent domain.
- Realising a particular design and implementation of an integrated agent architecture satisfying the requirements identified.
- Designing and implementing a highly instrumented and parametrized multi-agent simulation testbed with which to observe and analyse various aspects of agent-level problem solving, coordination, and behavioural ecology.

The resulting architecture — the TouringMachine agent architecture — integrates in a novel way a set of deliberative and non-deliberative agent control capabilities. These capabilities — which include situated action, attention focussing, planning, and causal reasoning via Belief/Desire/Intention modelling

— are distributed among three independent, concurrent, task-achieving control layers, the combination of which enables an agent to produce a range of reactive, goal-oriented, reflective, and predictive behaviours. Actions generated by the control layers are mediated through an enveloping subsumption-like control framework which is designed to select only those actions which are considered contextually appropriate. The combination of both fast non-deliberative control functions with preemptible, resource-bounded deliberative ones in such a layered control framework provides TouringMachines with *(i)* a guaranteed level of responsiveness, *(ii)* a degree of robustness to cope with a variety of exceptional events, *(iii)* the flexibility to adapt ongoing plans as and when required by changing circumstances in the environment, *(iv)* the ability to cope with events at several levels of spatio-temporal granularity, and *(v)* the ability to carry out several resource-constrained goals while at the same time coordinating their actions with other complex intentional agents.

9.2 Ideas for Future Work

Besides several potential weaknesses already pointed out earlier in the dissertation — for example, the use in layers $\mathcal{P}$ and $\mathcal{M}$ of a Focus of Attention Mechanism which relies on a relatively static focussing rule set (page 70), the assumption of linearity in layer $\mathcal{P}$'s Planner (page 83), and the assumption of noiseless sensory input (page 133) — there are a number of different ways in which the TouringMachine architecture could be extended and hopefully improved. Some of these are addressed below.

9.2.1 Explaining Behaviours

Through modelling of agent beliefs, desires, and intentions, TouringMachines are capable of generating explanations and predictions about their own and other agents' behaviours. In the current study, however, such modelling has been restricted to domains occupied by homogeneous agents: that is, agents with identical physical capabilities and essentially identical beliefs and desires — travel destination and deadline may differ but agents nevertheless hold the same set of prioritised goals. In other words, behaviour explanation in TouringMachines is centred on the recognition of agents' intentions, the assumption being that beliefs and desires are universally similar.

Clearly, in more complex environments populated by heterogeneous agents, TouringMachines ' explanatory powers would have to be enhanced to cope with a wider range of exceptional events. In particular, failure to explain an agent's behaviour solely through consideration of what its intentions might be may well be due to the fact that the observed agent possesses different beliefs, goals or perhaps even dissimilar physical capabilities or structure.[1] For example, a more sophisticated observer might be able to explain a "delinquent" agent's behaviour of running through a red traffic light by inferring that the agent does not possess `obey-regulations` as one of its homeostatic goals. Alternatively, it might explain its behaviour by inferring that it does possess said goal but that, from the observed agent's point of view, `obey-regulations` has a lower priority than its main achievement goal `reach-destination`. More sophisticated reasoning about agent behaviour would certainly benefit from a more sophisticated treatment of agent beliefs — in particular, representation [Kon83], default ascription [WB87], revision [DW90] — and agent desires — for example, values and likes [KR92], hedonic states [Gre87, page 308], and goal commitment [CL87].

As modelling capabilities are extended, however, agents will be faced with many more choices when it comes to explaining other agents' behaviour. Since TouringMachines are resource-bounded and must operate in real time, care will need to be taken if an appropriate level of responsiveness is still to be guaranteed. The completeness of models of other agents is a complex issue: more complete models of agents may be ineffective because they may require that an agent duplicate the processing of another node [BG88, page 27]. If an agent is to achieve any of its complex goals, its internal higher-level behavioural modules should not attempt to model every detail regarding its lower-level modules [Min86, page 169]. Likewise, no agent should attempt to model every detail of every other agent in order to explain or predict their behaviours. Knowing quite what to model, when to remodel, or how accurately selected events or entities should be modelled is likely to be environment- and task-related — as Hofstadter puts it:

> The intuition which is required for knowing when it makes sense to blur distinctions, to retry descriptions, to backtrack, to shift levels, and so forth, is something which probably comes only with much experience in thought in general. Thus it would be very hard to define heuristics for these crucial aspects of the program. [Hof79, page 661].

Further empirical evaluation of TouringMachines would likely help to shed

[1] By modelling other agents abstractly — in other words, solely in terms of their inferred propositional beliefs, desires, and intentions — TouringMachines could be said to be modelling each other at the knowledge level [New82, GN87, pages 313–320]: that is, approximately, without any reference to each other's physical details or structure. While this is probably appropriate for many domains, it may not be for those in which highly detailed predictions are needed in order to coordinate agent activity.

more light on these matters.

9.2.2 Controlling Inference

In order to guarantee an appropriate level of responsiveness in TouringMachines, strict limits are placed on the amount of deliberation which they are allowed to perform during any given execution cycle or timeslice. As described in previous chapters, operations such as focussing, planning and modelling are performed incrementally and have been designed in such a manner that they can be preempted when appropriate: that is, at the end of each timeslice or when the agent's per-timeslice computational resources become exhausted, whichever occurs first. However, since the distribution of computational resources among each of these control operations is made on the basis of a static, compile-time allocation scheme, TouringMachines will at times fail to make the best use of their limited resources. In particular, TouringMachines have no means of assessing, for a given situation, whether some further deliberation (and subsequent consumption of resources) should be favoured over the immediate execution of an action.

Of particular interest here are a number of decision-theoretic approaches to dealing with the twin problems of trading off deliberation versus execution and of characterising the quality of deliberatively inferred solutions vis-à-vis the time and resources available for arriving at such solutions. Notable examples for possible future consideration include incremental *anytime* algorithms as applied to deliberation scheduling and time-dependent planning schemes [DB86, BD90, DW91, pages 343–390], and utility-based metareasoning as a formal basis both for selecting among alternative action choices and for reasoning about the expected value of performing deliberative computations [RW89].[2] Doyle and Wellman also propose an interesting approach to incremental belief and plan revision which is rational, distributed, and which takes into account the utilities of the different deliberative computations involved [DW90].

---

[2]Note, however, that Hanks and Firby [HF90] have argued that it may ultimately be impossible at run-time and *in a principled way* to decide whether to act or deliberate further, largely because of the problem of characterising — in a negligible amount of time — the costs of the agent's ensuing benefits and lost opportunities. In the same paper, the authors also raise some interesting questions about the possibility of having deliberation and action take place simultaneously within the agent — an option not considered in other decision-theoretic architectures [HF90, page 67].

9.2.3 Adaptive Behaviour

According to Hofstadter, the flexibility of intelligence comes from having an enormous number of rules, and levels of rules. The reason for this, he suggests, is that, in real life, agents will have to deal with countless situations of completely different types. Depending on the type of situation an agent could find itself in, he further suggests [Hof79, page 27], it might be the case that stereotyped responses using "just plain" rules[3] would be appropriate. In other situations — those which are mixtures or combinations of stereotyped situations — the agent would probably require rules for deciding which of the "just plain" rules to apply.[4] In yet other situations — those which cannot be readily classified or which are *novel* — there may need to exist rules for inventing or *learning* new rules.

A key ingredient of intelligent behaviour in an agent must surely, then, include the ability to *adapt* to novel situations and to learn new behaviours. Learning requires that an agent be able to make changes to its internal structure so as to improve some metric on its long-term future performance according to some fixed performance criterion [Rus89]. In this particular sense TouringMachines can at present be considered *pre-intelligent*: in response to environmental change, TouringMachines can, by changing their intentions, dynamically alter their internal control structure; however, they do so without regard to any metric on their long-term future performance.

A TouringMachine agent is currently programmed via a set of internal behavioural parameters. By virtue of being able to sense its environment and monitor the effects of both its own and others' actions, a TouringMachine effectively possesses a feedback loop between itself and its surroundings. For a TouringMachine to be considered *adaptive*, it would require a secondary feedback loop, perhaps in the form of a fourth control layer $\mathcal{L}$(see Figure 9.1), which would enable the agent to analyse the performance exhibited by its three existing control layers and thus make appropriate parameter value changes which optimise its performance according to some particular performance criterion. Movement through the agent's parameter space would likely be achieved via a standard gradient-descent search [DW91, page 399]. Possible criteria for assessing an agent's performance include, among others, the temporal delay in accomplishing its planned task, the utilisation of computational resources throughout its various control layers, the frequency with which modelling discrepancies arise, and the rate at which near-miss collisions involving the agent occur.[5]

---

[3]Compare with a TouringMachine's reactive (layer $\mathcal{R}$) rules.

[4]Compare with a TouringMachine's planning (layer $\mathcal{P}$) and modelling (layer $\mathcal{M}$) functions.

[5]If several performance criteria were to be used simultaneously, they could be combined in

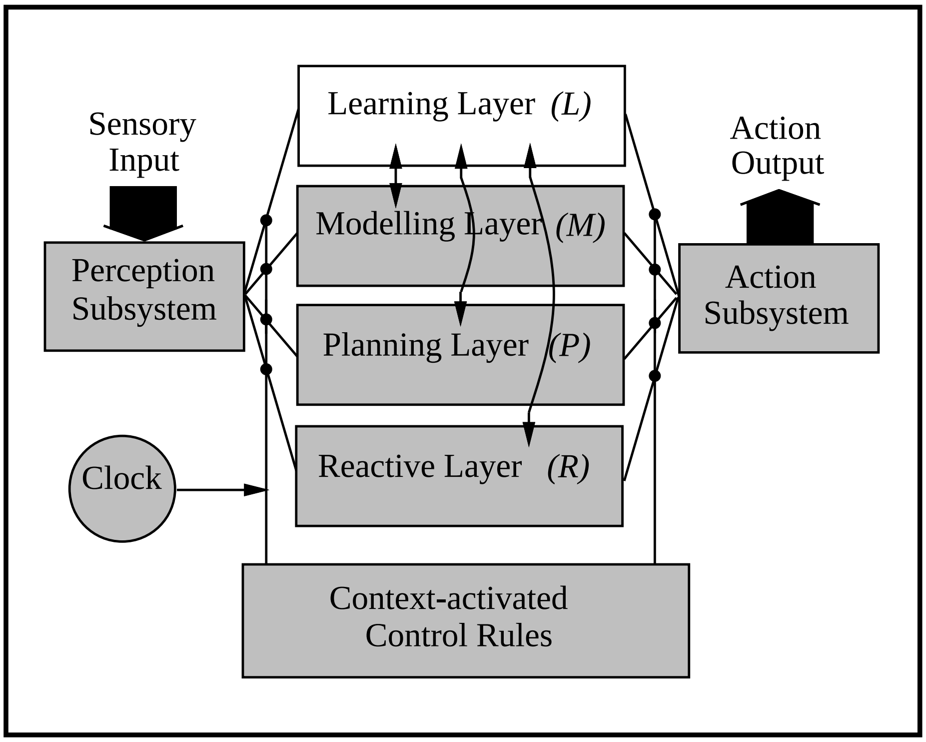

Figure 9.1: The TouringMachine architecture would benefit from the addition of a fourth control module for learning — layer $\mathcal{L}$. This module would be charged with assessing the agent's performance (according to some pre-specified criterion) and fine-tuning the other three layers' internal parameters so that the agent's performance might improve over time.

In addition to learning by adjusting internal parameters — also known as *reinforcement learning* [DW91, page 394] — TouringMachines would likely benefit from a second style of learning, often referred to as *performance learning*. This can take the form of off-line knowledge compilation, for example from plans to situation-action rules [DR91], or from symbolic goal-reduction rules to executable condition-action pairs [Kae91]. Similarly, various approaches at integrating explanation-based learning capabilities in agents can also be classified as performance learning. For example, *chunking* to compile impasse resolution procedures in SOAR [LR90], derivational analogy to acquire domain-specific control rules in PRODIGY [CEG$^+$91], as well as *caching* of stimulus response rules in Theo [Mit90]. Russell [Rus89] also proposes a decision-theoretic framework for knowledge compilation within which each of the above approaches can be formally described. Besides the potential for making agents more reactive — and thus more run-time efficient — performance learning can help to overcome the restriction that all of the agent's

a weighted vector sum.

stimulus-response pairs be specified by the programmer in advance, which, for example, is required in the architectures of Agre and Chapman [AC87] and Schoppers [Sch87].

9.2.4 Social Agency

Most of the AI research carried out to date has been *asocial* — it has concerned itself with issues about knowledge, reasoning, and acting in *individual* systems or agents rather than in groups, organisations or *societies* of these. TouringMachines are similarly asocial: the emphasis of the present research has been on the architectural requirements of individual agents to carry out tasks in domains where there are no group or shared tasks and where individual cooperation, rationality, and common inter-agent semantics could be assumed.

As the technology used to develop integrated agent architectures matures, these intelligent systems will increasingly become embedded in complex organisations comprising humans as well as other intelligent computational agents. Increasingly, thus, there will be a growing incentive to study more closely some of the social and organisational dimensions of intelligent agency. In particular, as Gasser [Gas91] points out, there will be a requirement to treat the existence of *multiple* agents as a fundamental category, to address the tension between local perspectives of knowledge and action and their social counterparts (for example, common knowledge [Dav90], ethics [Asi50]), to assume agents employ multiple representations of knowledge and hold multiple, disparate perspectives of their shared tasks, and to account for joint courses of action which are robust over time to failure and inconsistency.

Many important ideas concerning intelligent agency have begun to emerge from such fields as economics, sociology, cognitive science, ecology, and ethology, among others. Particularly interesting avenues for future research include the investigation of sophisticated behaviours which emerge from the interactions of groups of relatively simple behaviours [AD90] or agents [Min86], the role of joint intention and inter-agent communication in establishing joint commitments toward common goals [LCN90], and the use of market-based systems — for example, *computational ecologies* [HH88] or *agoric open systems* [MD88] — for enabling the robust integration of and flexible resource allocation among asynchronous, heterogeneous, resource-bounded computational agents.

9.3 Conclusions

The thesis of this dissertation is that it is both desirable and feasible to combine deliberative and non-deliberative control functions in order to obtain effective, robust, and flexible behaviour from autonomous task-achieving agents which are intended to operate in complex domains. A secondary hypothesis which has been investigated in this dissertation is the claim that establishing an appropriate balance between reasoning (deliberative control) and acting (non-deliberative control) depends heavily on characteristics of the task environments in which the agents are intended to operate.

The survey of existing intelligent agent architectures in Chapter 2 indicates that many approaches have restrictively addressed isolated behavioural requirements of agents — for example, the need to react to unforeseen events or the need to plan complex tasks. Taking into consideration the fundamental design trade-off between an agent's representational power (the generality and flexibility of its behaviour, the ease with which the designer can program, test, and debug it) and its computational tractability (its run-time efficiency), the analysis of the numerous functional and behavioural requirements for autonomous agency in the chosen complex domain suggests that employing a hybrid architecture — one combining a selection of both deliberative and non-deliberative functions — is likely to prove the most promising approach.

The TouringMachine agent architecture, one possible solution to such a hybrid control approach, is based on a number of existing traditional and non-traditional AI techniques which have been suitably modified and integrated in a novel manner. The architecture, thus, grows out of the modification and extension — rather than the rejection — of classical AI techniques such as planning and model-based reasoning. As such, this dissertation should help to improve our understanding of some of the key functional and behavioural requirements faced by *planning* agents which are to operate satisficingly in complex environments.

The TouringMachine architecture is centred on a number of modular, independent, task-achieving control layers which, apart from ensuring a high degree of operational concurrency, enable agents to handle multiple goals, carry out resource-constrained tasks, coordinate their activities with other rational agents in an effective manner, and cope with a range of events at differing levels of granularity. Because of the real-time constraints typically imposed on agents embedded in dynamic environments, an account is taken of TouringMachines ' limited computational resources in order to guarantee an upper bound on the latency of such operations as sensing, attention focussing, planning, and world modelling. The result is an architecture which can produce a number of reactive, goal-directed, reflective, and predictive behaviours, as and when dictated by the agent's mental state and environmental context.

The inherent complexity of integrated agent architectures, combined with the relative immaturity of the field, makes evaluation of different agent designs a difficult and drawn out process. The approach taken in this dissertation has been to construct as complete a prototype implementation as time and resources would permit, and then to evaluate the feasibility of the architecture in order to identify its various strengths and weaknesses. This was done through a number of empirical evaluations using a purpose-built, feature-rich, instrumented simulation testbed. A complete evaluation, and in particular, the construction of a detailed causal model accounting for a TouringMachine's full behavioural ecology over a wide range of task domains, would take a far greater amount of time than was available for the dissertation.

The results of the evaluations in Chapter 8 show that the TouringMachine architecture is feasible and that, suitably configured, can endow rational autonomous agents with appropriate levels of effective, robust, and flexible control for handling a number of pre-programmed, dynamic, and also hypothetical events. While the TouringMachine architecture is inherently capable of producing a range of deliberative and non-deliberative behaviours, the results also show that establishing an *ideal* agent parametrization — one which is intended to produce a particular type of desired behaviour — depends very strongly on the agent's particular task constraints and environmental circumstances. In addition, while the need to react quickly to unexpected events is universally considered a desirable behaviour for autonomous agents, the evaluation of TouringMachines also suggests that, at least in some circumstances, the ability to perform some amount of causal reasoning about other agents' intentions and desires can prove very useful in promoting effective task coordination between agents.

Experience in designing, implementing, and testing the TouringMachine architecture also shows that dividing control among several modular, task-achieving layers can facilitate incremental testing and debugging of different agent configurations. In particular, because TouringMachines' control layers are independently connected to their sensors and effectors, agents could be tested very early on during the course of the research: indeed, as soon as their reactive layers had been implemented. As the other control layers were completed and put in place so it became possible to analyse and debug TouringMachines which, incrementally, were able to handle more complex tasks and exhibit a wider range of behaviours.

The strengths of the TouringMachine architecture and implementation lie in the ability to produce — by means of a carefully designed and integrated collection of deliberative and non-deliberative control techniques — a diverse range of intelligent autonomous behaviours. Having given empirical evaluation a high priority from the outset of their design, TouringMachines have been extensively parametrized in order to facilitate detailed study via a purpose-built simulation testbed. Enabling rapid and effective analysis of a range of different agent and environment configurations, the combination of the TouringMachine architecture and testbed should prove a powerful platform for studying a number of important issues concerning autonomous agency, particularly as research into integrated agent architecture matures and suitable benchmark tests begin to emerge.

In its present form, the architecture also exhibits a number of weaknesses. In particular, the inability to reason in any sophisticated way about the utilisation and distribution of computational resources among their various control layers prevents TouringMachines from explicitly addressing at run-time tradeoffs between needing to deliberate further and needing to act. Similarly, the inability to reason explicitly about internal parameter settings and about how different parameter settings might promote or hinder certain behaviours in different environmental contexts prevents TouringMachines from learning from their past mistakes and from intelligently adapting to changing world conditions. These and several other limitations have been discussed in more detail in previous sections.

The result is a novel architectural design which can successfully produce a range of useful behaviours required of embedded, rational, autonomous agents operating in complex multi-agent domains. Implemented and evaluated in a richly parametrized multi-agent testbed, TouringMachines should serve to advance our understanding of many of the wider issues concerning the practical construction of intelligent, autonomous, integrated systems — systems which, without a doubt, will necessarily play a vital role in the forging of tomorrow's large-scale, evolving computational networks and decentralised information processing systems.

Appendix A

TouringWorld Scenario Grammar

The syntax of the TouringWorld Scenario Specification Language, SSL, is given below in Extended Backus-Naur Form. Non-terminals appear as italicised terms (for example, *scenario-level-declarations*). Terminals include scenario parameter names (for example, **ScenarioName**), base programming lang-
uage types (for example, `POSITIVE FLOAT`), as well as quoted strings. The various syntax rules have been split into sections corresponding to the three non-terminals appearing in the body of the top-most rule:

scenario-specification ::=
    scenario-level-declarations ';;'
    environment-level-declarations ';;'
    entity-level-declarations '.'

A.1 Scenario-level Declarations

scenario-level-declarations ::=
    ScenarioName '=' *scenario-name*
    ScenarioIterations '=' *scenario-iterations*
    ScenarioSuspension '=' *start-time* ',' *suspension-rate*
    TerminationCriterion '=' *termination-criterion*
    SaveScript '=' *save-script-status* ',' *save-script-file*
    UserPlayback '=' *playback-status* ',' *playback-file*
    [*window-definitions*]
    [*graph-plot-declarations*]

[*graphics-declarations*]
[*recordable-parameter-declarations*]
[*alterable-parameter-declarations*]
[*parameter-window-declarations*]
[*trace-declarations*]

window-definitions ::=
window-definition { ';' *window-definition* }
graph-plot-declarations ::=
PlotMode '=' *plot-mode*
PlotStyle '=' *plot-style*
PlotCurveLabel '=' *plot-label*
PlotXAxis '=' *plot-parameter* ',' *plot-label*
PlotXOrigin '=' *plot-origin*
PlotXRange '=' *plot-range*
PlotXDisplayIncrement '=' *plot-display-increment*
PlotXAxisMarkerIncrement '=' *plot-axis-marker-increment*
PlotXAxisMarkerSize '=' *plot-axis-marker-size*
PlotYAxis '=' *plot-parameter* ',' *plot-label*
PlotYOrigin '=' *plot-origin*
PlotYRange '=' *plot-range*
PlotYDisplayIncrement '=' *plot-display-increment*
PlotYAxisMarkerIncrement '=' *plot-axis-marker-increment*
PlotYAxisMarkerSize '=' *plot-axis-marker-size*
graphics-declarations ::=
GraphicsWindowCoordinates '=' *graphics-window-coordinates*
ScrollGraphics '=' *scroll-graphics*
[*agent-shape-declarations*]
[*front-sensing-arc-declarations*]
[*rear-sensing-arc-declarations*]
recordable-parameter-declarations ::=
recordable-parameter-declaration { ';' *recordable-parameter-declaration* }
alterable-parameter-declarations ::=
alterable-parameter-declaration { ';' *alterable-parameter-declaration* }
parameters-window-declarations ::=
parameters-window-declaration { ';' *parameters-window-declaration* }
trace-declarations ::=
trace-declaration { ';' *trace-declaration* }

window-definition ::=
WindowName '=' *window-name*

WindowProcess '=' *window-process*
WindowScreenPosition '=' *window-screen-x* ',' *window-screen-y*
WindowDimensions '=' *window-height* ',' *window-width*
WindowMaxLines '=' *window-max-lines*
WindowContrast '=' *window-contrast*
WindowCursor '=' *window-cursor*
scroll-graphics ::=
'rate:' *graphics-increment-rate* ',amount:' *graphics-increment-amount*
agent-shape-declarations ::=
agent-shape-declaration { ';' *agent-shape-declaration* }
front-sensing-arc-declarations ::=
front-sensing-arc-declaration { ';' *front-sensing-arc-declaration* }
rear-sensing-arc-declarations ::=
rear-sensing-arc-declaration { ';' *rear-sensing-arc-declaration* }
recordable-parameter-declaration ::=
RecordableParameter '=' *recordable-parameter* ',scope:'
parameter-scope ',style:' *parameter-style*
alterable-parameter-declaration ::=
AlterableParameter '=' *alterable-parameter* ',scope:' *parameter-scope*
',change:' *parameter-change* ',amount:' *parameter-change-amount*
',limit:' *parameter-bound*
parameters-window-declaration ::=
ParmsWindowEntry '=' *parms-window-entry*
trace-declaration ::=
TraceParameter '=' *trace-parameter* ',' *parameter-scope*

agent-shape-declaration ::=
AgentGraphicsShape '=' *agent-name* ',' *agent-shape*
front-sensing-arc-declaration ::=
DrawForwardSensingArc '=' *agent-name* ',' *start-time*
rear-sensing-arc-declaration ::=
DrawRearSensingArc '=' *agent-name* ',' *start-time*
recordable-parameter ::=
recordable-tmw-parameter ',' *user-code-pointer*
alterable-parameter ::=
alterable-tmw-parameter ',' *user-code-pointer*

scenario-name ::= IDENTIFIER
scenario-iterations ::= NATURAL NUMBER
start-time ::= *world-time*
suspension-rate ::= POSITIVE INTEGER

termination-criterion ::=
 `'first-collision'` | `'terminated:'` *agent-name* |
 `'duration:'` `POSITIVE FLOAT` | `'all-terminated'`
save-script-status ::= `'on'` | `'off'`
save-script-file ::= `FILE IDENTIFIER`
playback-status ::= `'record'` | `'playback'` | `'off'`
playback-file ::= `FILE IDENTIFIER`

window-name ::= `IDENTIFIER`
window-process ::=
 `'environment'` | `'graphics'` | `'graph-plot'` |
 `'user-agent'` | `'parameters'` | *agent-name*
window-screen-x ::= `POSITIVE INTEGER`
window-screen-y ::= `POSITIVE INTEGER`
window-height ::= `POSITIVE INTEGER`
window-width ::= `POSITIVE INTEGER`
window-max-lines ::= `NATURAL NUMBER`
window-contrast ::= `'positive'` | `'negative'`
window-cursor ::= `X FONT CURSOR`[1]

plot-mode ::= `'incremental'` | `'non-incremental'`
plot-style ::= `'dot'` | `'cross'` | `'circle'`
plot-label ::= `'X:'` `NATURAL NUMBER` `',Y:'` `NATURAL NUMBER` `',label:'` `STRING`
plot-parameter ::= *recordable-parameter*
plot-origin ::= `POSITIVE FLOAT`
plot-range ::= `NATURAL NUMBER`
plot-display-increment ::= `NATURAL NUMBER`
plot-axis-marker-increment ::= `NATURAL NUMBER`
plot-axis-marker-size ::= `NATURAL NUMBER`

graphics-window-coordinates ::=
 `'leftX:'` `NATURAL NUMBER,` `',rightX:'` `NATURAL NUMBER`
 `',topY:'` `NATURAL NUMBER,` `',bottomY:'` `NATURAL NUMBER`
graphics-increment-rate ::= `POSITIVE FLOAT`
graphics-increment-amount ::= `NATURAL NUMBER`
agent-shape ::= `'circle'` | `'triangle'` | `'enterprise'`

parameter-scope ::= `'environment'` | *agent-name*
parameter-style ::= `'incremental'` | `'non-incremental'`

[1] Available cursor shapes are listed in Appendix C of the XWIP Manual [Kim90, page 51].

recordable-tmw-parameter ::=
`'schemas-retrieved'` | `'schemas-discarded'` | `'completed-subtasks'` |
`'total-actions-effected'` | `'reactive-rules-fired'` |
`'entities-sensed'` | `'focussed-entities'` |
`'focussing-predicates-used'` | `'entities-modelled'` |
`'resources-spent(sensing)'` | `'resources-spent(focussing)'` |
`'resources-spent(planner)'` | `'resources-spent(modeller)'` |
`'conflicts-detected'` | `'conflicts-resolved'` | `'total-agents'` |
`'total-objects'` | `'layer-suppressions'` | `'layer-censorings'` |
`'task-delay'` | `'failed-tasks'` | `'successful-tasks'` |
`'average-speed'` | `'agent-collisions'` | `'final-distance-to-target'` |
`'average-resources(sensing)'` | `'average-resources(focussing)'` |
`'average-resources(planner)'` | `'average-resources(modeller)'` |
`'average-utilisation(sensing)'` | `'average-utilisation(focussing)'` |
`'average-utilisation(planner)'` | `'average-utilisation(modeller)'` |
`'average-focussed-entities'` | `'average-sensed-entities'` |
`'average-modelled-entities'` | `'actions-effected-by(layer-R)'` |
`'actions-effected-by(layer-P)'` | `'actions-effected-by(layer-M)'`
user-code-pointer ::= `PROLOG CLAUSE`

parameter-change ::= `'increment'` | `'decrement'`
parameter-change-amount ::= `FLOAT`
parameter-bound ::= `FLOAT`
alterable-tmw-parameter ::=
`'world-time-increment'` | `'entity-size'` | `'environment-factor'` |
`'task(travel-time)'` | `'task(light-rate)'` | `'schema-retrieval-cost'` |
`'schema-placing-cost'` | `'max-resources(sensing)'` |
`'max-resources(focussing)'` | `'max-resources(planner)'` |
`'max-resources(modeller)'` | `'reaction(kerb-threshold)'` |
`'reaction(frontal-agent-threshold)'` | `'reaction(rear-agent-threshold)'` |
`'reaction(obstacle-threshold)'` | `'reaction(lane-marking-threshold)'` |
`'forward-sensing-range'` | `'rear-sensing-range'` |
`'forward-sensing-arc'` | `'rear-sensing-arc'` | `'sensing-rate'` |
`'modelling-rate'` | `'model-discard-time'` | `'model-bounds(location)'` |
`'model-bounds(speed)'` | `'model-bounds(acceleration)'` |
`'model-bounds(orientation)'` | `'conflict-detection-horizon'` |
`'conflict-resolution-depth'`

parms-window-entry ::= *recordable-parameter*

trace-parameter ::=
`'time'` | `'final-collisions'` | `'final-agents'` | `'agent'` |
`'environment-agent'` | `'terminated'` | `'effectors'` |
`'message-buffer'` | `'sensors'` | `'visible-entities'` |
`'occluded-entities'` | `'focus-of-attention'` | `'focussed-entities'` |
`'reactive-layer'` | `'planning-layer'` | `'task-handler'` |
`'planner-step'` | `'planner-operation'` | `'modelling-layer'` |
`'model-retrieval'` | `'model-discrepancy'` | `'model-resolution'` |
`'model-expectation'` | `'messages-sent'` | `'messages-received'` |
`'layer-suppression'` | `'layer-censoring'`

A.2 Environment-level Declarations

environment-level-declarations ::=
InitialWorldTime `'='` *world-time*
WorldTimeIncrement `'='` *world-time-increment*
FogFactor `'='` *environment-factor*
RainFactor `'='` *environment-factor*
calculation-truncations

calculation-truncations ::=
DistanceTruncation `'='` *truncation-factor*
SpeedTruncation `'='` *truncation-factor*
AccelerationTruncation `'='` *truncation-factor*
AngleTruncation `'='` *truncation-factor*

world-time ::= `POSITIVE FLOAT`
world-time-increment ::= `POSITIVE FLOAT`
environment-factor ::= `POSITIVE FLOAT` $\leq$ `1.0`
truncation-factor ::= `POSITIVE FLOAT`

A.3 Entity-level Declarations

entity-level-declarations ::=
entity-definition { `';'` *entity-definition* }

entity-definition ::=
agent-definition | *environment-agent-definition* |

user-agent-definition | *object-definition*

agent-definition ::=

AgentName '=' *agent-name*

AgentProcessType '=' *agent-process-type*

AgentVector '=' *agent-vector*

EntitySize '=' *entity-radius*

agent-capability-declarations

control-rule-declarations

sensor-declarations

focus-of-attention-declarations

layer-R-declarations

layer-P-declarations

layer-M-declarations

environment-agent-definition ::=

EnvironmentAgentName '=' *agent-name*

AgentProcessType '=' *agent-process-type*

EnvironmentAgentVector '=' *environment-agent-vector*

EnvironmentAgentAttributes '=' *environment-agent-attributes*

EntitySize '=' *entity-radius*

layer-P-declarations

user-agent-definition ::=

UserAccelerationIncrement '=' *user-agent-increment*

UserDecelerationIncrement '=' *user-agent-decrement*

UserTurnLeftIncrement '=' *user-agent-decrement*

UserTurnRightIncrement '=' *user-agent-increment*

agent-vector

agent-capability-declarations

object-definition ::=

ObjectName '=' *object-name*

ObjectType '=' *object-type*

ObjectLocation '=' *world-x-location* ',' *world-y-location*

ObjectAttributes '=' *object-attributes*

EntitySize '=' *entity-radius*

agent-vector ::=

start-time ',' *world-x-location* ',' *world-y-location* ','
agent-speed ',' *agent-acceleration* ',' *agent-orientation* ','
agent-communications

agent-capability-declarations ::=

MaxSpeed '=' *agent-speed*

MaxAcceleration '=' *agent-acceleration*

MaxDeceleration '=' *agent-deceleration*
MaxTurningRate '=' *agent-orientation*
CommsDeviceStatus '=' *comms-device-status*
control-rule-declarations ::=
censor-control-rule { ',' *censor-control-rule* }
suppressor-control-rule { ',' *suppressor-control-rule* }
sensor-declarations ::=
SensingAlgorithm '=' *sensing-algorithm* ',cost:' *resources*
ForwardSensingRange '=' *sensing-range*
RearSensingRange '=' *sensing-range*
ForwardSensingArc '=' *sensing-arc*
RearSensingArc '=' *sensing-arc*
SensingRate '=' *sensing-rate*
focus-of-attention-declarations ::=
FocussingRules '=' *focussing-rules*
FocussingResources '=' *resources*
FocussingEntityCost '=' *resources*
FocussingFlagCost '=' *resources*
layer-R-declarations ::=
initial-reactive-rules
KerbThreshold '=' *separation*
KerbAvoidanceAngle '=' *agent-orientation*
WallThreshold '=' *separation*
WallAvoidanceSpeed '=' *agent-speed*
FrontalAgentThreshold '=' *separation*
FrontalAvoidanceSpeed '=' *agent-speed*
RearAgentThreshold '=' *separation*
RearAvoidanceSpeed '=' *agent-speed*
ObstacleThreshold '=' *separation*
ObstacleAvoidanceAngle '=' *agent-angle*
LaneMarkingThreshold '=' *separation*
LaneMarkingAvoidanceAngle '=' *agent-angle*
layer-P-declarations ::=
PlannerAlgorithm '=' *planner-algorithm*
PlannerResources '=' *resources*
PlannerTask '=' *planner-task*
SchemaRetrievalCost '=' *resources*
SchemaPlacingCost '=' *resources*
HasSchemas '=' *schema-names*
topological-world-map
layer-M-declarations ::=

ModellerAlgorithm '=' *modeller-algorithm*
ModellerResources '=' *resources*
ModelRetrievalCost '=' *resources*
ModellerFlagCost '=' *resources*
ConflictDetectionHorizon '=' *conflict-detection-horizon*
ConflictResolutionDepth '=' *conflict-resolution-depth*
ModellingRate '=' *modelling-rate*
ModelDiscardAfterTime '=' *model-discard-after-time*
ModelLocationBounds '=' *model-bounds*
ModelSpeedBounds '=' *model-bounds*
ModelAccelerationBounds '=' *model-bounds*
ModelOrientationBounds '=' *model-bounds*
HasModelTemplates '=' *model-template-names*
theorist-kbase-rules
space-time-projection-functions
conflict-resolution-rules

environment-agent-vector ::=
start-time ',' *world-x-location* ',' *world-y-location* ','
environment-agent-type ',' *stop-time* ',' *environment-communication*
environment-agent-attributes ::= *traffic-light-attributes* | '[]'
object-attributes ::=
wall-attributes | *lane-marking-attributes* | *sign-attributes* | '[]'

agent-communications ::=
'[' *agent-communication* { ',' *agent-communication* } ']'
censor-control-rule ::=
CensorControlRule '=' *rule-name* ': if '
censor-condition { ' and ' *censor-condition* }
' then ' *censor-action* { ' and ' *censor-action* }
suppressor-control-rule ::=
SuppressorControlRule '=' *rule-name* ': if '
suppressor-condition { ' and ' *suppressor-condition* }
' then ' *suppressor-action* { ' and ' *suppressor-action* }
focussing-rules ::=
focussing-rule { ',' *focussing-rule* }
initial-reactive-rules ::=
reactive-rule { ',' *reactive-rule* }
planner-task ::= *plan-route-task* | *light-task* | *rain-task* | *fog-task*
schema-names ::=
schema-name { ',' *schema-name* }
topological-world-map ::=

```
        path-definitions
        junction-definitions
model-template-names ::=
        model-template-name { ',' model-template-name }
theorist-kbase-rules ::=
        theorist-kbase-rule { ',' theorist-kbase-rule }
space-time-projection-functions ::=
        space-time-projection-function { ',' space-time-projection-function }
conflict-resolution-rules ::=
        conflict-resolution-rule { ',' conflict-resolution-rule }
traffic-light-attributes ::=
        'size:'  POSITIVE FLOAT ',orientation:' traffic-light-orientation
wall-attributes ::= 'length:'  POSITIVE FLOAT ',type:' wall-type
lane-marking-attributes ::= 'path:' path-name ',type:' lane-marking-type
sign-attributes ::=
        'text:'  STRING ',path:' path-name ',junction:' junction-name

censor-condition ::= 'entity(' entity-name ') ∈ Perception-Buffer'
censor-action ::=
        'remove-sensory-record(' layer-name ', entity(' entity-name '))'
suppressor-condition ::= action-command-spec | agent-state-spec
suppressor-action ::=
        'remove-action-command(' layer-name ',' agent-action ')'
focussing-rule ::=
        focussing-predicate { '&' focussing-predicate }
plan-route-task ::=
        '(plan-a-route, [[agent,' agent-name '],'
                        '[final-destination,' destination '],'
                        '[travel-time,' travel-time '],'
                        '[within-distance,' distance ']])'
light-task ::=
        '(light, [[agent,' agent-name '],'
                 '[interval,' time-interval '],'
                 '[amber-time,' time-interval '],'
                 '[colours, [' colour ',' colour ',' colour ']]])'
fog-task ::= '(fog, [[visibility-factor,' environment-factor ']])'
rain-task ::= '(rain, [[friction-factor,' environment-factor ']])'
path-definitions ::=
        path-definition { ';' path-definition }
junction-definitions ::=
        junction-definition { ';' junction-definition }
```

theorist-kbase-rule ::=
    TheoristKBaseEntry '=' *theorist-kbase-entry*
space-time-projection-function ::=
    SpaceTimeProjection '=' *configuration* ',' *intention* ','
    timeperiod ':' *projection*
conflict-resolution-rule ::=
    ConflictResolutionRule '=' *conflict-type* ',' *conflict-goal* ','
    conflict-resolution

path-definition ::=
    TouringWorldPath '=' *path-name* ',' *path-width* ',' *junction-name* ','
    junction-name
junction-definition ::=
    TouringWorldJunction '=' *junction-name* ',' *world-x-location* ','
    world-y-location
configuration ::=
    '((' *world-x-location* ',' *world-y-location*), *agent-speed* ','
    agent-acceleration ',' *agent-orientation* ',' *agent-communications* '))'
intention ::= *schema-name*
projection ::=
    '[' *agent-path* { ',' *agent-path* } ']'
conflict-goal ::= *conflicting-goal* | *self-model-conflicting-goal*
conflict-resolution ::= *action-specification* | *task-specification*

agent-name ::= `IDENTIFIER`
agent-process-type ::= `'internal'` | `'external'`
agent-speed ::= `POSITIVE FLOAT`
agent-acceleration ::= `POSITIVE FLOAT`
agent-orientation ::= `POSITIVE FLOAT` $\leq$ `360.0`
entity-radius ::= `POSITIVE FLOAT`
agent-deceleration ::= `NEGATIVE FLOAT`
agent-communication ::=
    `'signal-left'` | `'signal-right'` | `'flash-headlights'` |
    `'honk-horn'` | `'braking'` | `'hazard-lights'` | `'fog-lights'`
comms-device-status ::= `'ok'` | `'faulty'`
rule-name ::= `IDENTIFIER`
entity-name ::= *agent-name* | *object-name*
layer-name ::= `'layer-R'` | `'layer-P'` | `'layer-M'`
action-command-spec ::=
    `'action-command('` *action-source* ',' *agent-action* `')` $\in$ `Action-Buffer'`

agent-state-spec ::= `'current-intention('` *intention* `')'`
sensing-algorithm ::=
    `'all'` | `'all-but-covered'` | `'all-ahead'` |
    `'all-but-covered-ahead'` | `'all-but-occluded'` |
    `'all-but-covered-and-occluded'` | `'all-ahead-but-occluded'` |
    `'all-ahead-but-covered-and-occluded'` | *user-code-pointer*
resources ::= NATURAL NUMBER
sensing-range ::= POSITIVE FLOAT
sensing-arc ::= POSITIVE FLOAT $\leq$ 360.0
sensing-rate ::= NATURAL NUMBER
focussing-predicate ::= FOCUSSING PREDICATE[2] | *user-code-pointer*

reactive-rule ::=
    `'rule-1: kerb-avoidance'` | `'rule-2: wall-avoidance'` |
    `'rule-3: front-agent-avoidance'` | `'rule-4: rear-agent-avoidance'` |
    `'rule-5: obstacle-avoidance'` | `'rule-6: lane-marking-avoidance'` |
    `'rule-7: direction-zeroing'` | *user-code-pointer*
separation ::= POSITIVE FLOAT
planner-algorithm ::= `'layer-P'` | *user-code-pointer*
schema-name ::=
    `'plan-a-route'` | `'determine-route'` | `'determine-cruise-speed'` |
    `'follow-route'` | `'attain-speed'` | `'go-to-location'` | `'query-speed'` |
    `'turn-to-target'` | `'stop-over-distance'` | `'query-current-location'` |
    `'query-object-location'` | `'find-nearest-kerb'` | `'follow-path'` |
    `'find-nearest-lane-marking'` | `'query-path'` | `'select-shortest-route'` |
    `'calculate-route-distance'` | `'get-next-junction'` |
    `'follow-a-path'` | `'repeat-follow-route'` | `'query-junction-location'` |
    `'calculate-speed'` | `'accelerate'` | `'determine-speed-acceleration'` |
    `'choose-acceleration-rate'` | `'decelerate'` | `'choose-deceleration-rate'` |
    `'determine-acceleration-period'` | `'calculate-stopping-distance'` |
    `'calculate-turning-angle'` | `'turn-to-angle'` | `'drive-along-path'`
    `'signal-left'` | `'signal-right'` | `'flash-headlights'` | `'honk-horn'` |
    `'hazard-lights'` | `'fog-lights'` | `'stop-at-light'` |
    `'traffic-light-stop'` | `'poll-traffic-light'` | `'wait-for-change'` |
    `'small-accelerate'` | `'stop-at-junction'` | `'poll-junction-agent'` |
    `'nearest-entry'` | `'stop-behind-agent'` | `'halt-over-distance'` |
    `'poll-stopped-agent'` | `'start-overtake'` | `'finish-overtake'` |
    `'light'` | `'light-on-off-times'` | `'light-status'` | `'rain'` |
    `'rain-on'` | `'rain-off'` | `'fog'` | `'fog-on'` |
    `'fog-off'` | *user-code-pointer*

[2]TouringWorld focussing predicates are listed in Chapter 5, Figure 5.3.

destination ::= *object-name* | *junction-name*
travel-time ::=
    `'eq'` POSITIVE FLOAT | `'within'` POSITIVE FLOAT `'of'` POSITIVE FLOAT |
    `'lt'` POSITIVE FLOAT | `'gt'` POSITIVE FLOAT
distance ::= POSITIVE FLOAT
time-interval ::= POSITIVE FLOAT
colour ::= `'red'` | `'green'` | `'amber'`
path-name ::= IDENTIFIER
path-width ::= POSITIVE FLOAT
junction-name ::= IDENTIFIER
world-x-location ::= POSITIVE FLOAT
world-y-location ::= POSITIVE FLOAT
modeller-algorithm ::= `'layer-M'` | *user-code-pointer*
conflict-detection-horizon ::= NATURAL NUMBER
conflict-resolution-depth ::= NATURAL NUMBER
modelling-rate ::= NATURAL NUMBER
model-discard-after-time ::= POSITIVE FLOAT
model-bounds ::= `'+/-'` POSITIVE FLOAT
model-template-name ::=
    `'self'` | `'touring-machine'` | `'traffic-lights'` |
    `'fog'` | `'rain'` | *user-code-pointer*
theorist-kbase-entry ::=
    `'fact:'` CLAUSE[3] `';'` | `'default:'` *schema-name* `' '` CLAUSE `';'`
agent-path ::= `'('` *configuration* `','` *timeperiod* `')'`
timeperiod ::= POSITIVE FLOAT
conflict-type ::=
    `'junction-collision-and-no-right-of-way'` |
    `'junction-collision-and-right-of-way'` |
    `'frontal-collision-and-way-is-clear'` |
    `'frontal-collision-and-way-is-not-clear'` |
    `'red-traffic-light'` | `'other-traffic-light-colour'` |
    `'unexpected-signalling'` | `'unexpected-braking'` |
    `'unexpected-speed'` | `'unexpected-orientation'` |
    `'new-entity'` | *user-code-pointer*
conflicting-goal ::=
    `'avoid-collisions'` | `'obey-regulations'` | *self-model-conflicting-goal*
self-model-conflicting-goal ::= `'reach-destination'`
action-specification ::= `'['` *agent-action* `','` *action-arguments* `']'`
task-specification ::= `'['` *intention* `','` *intention-arguments* `']'`

[3] A CLAUSE is any formula in first order clausal form logic [PGA86].

environment-agent-type ::= `'fog'` | `'rain'` | `'traffic-light'`
stop-time ::= POSITIVE FLOAT
environment-communication ::=
  `'rain'` | `'fog'` | *colour*
traffic-light-orientation ::=
  `'bottom-top'` | `'top-bottom'` | `'left-right'` | `'right-left'`

user-agent-increment ::= POSITIVE FLOAT
user-agent-decrement ::= NEGATIVE FLOAT

object-name ::= IDENTIFIER
object-type ::=
  `'obstacle'` | `'wall'` | `'kerb'` | `'information-sign'` |
  `'entry'` | `'exit'` | `'lane-marking'`
wall-type ::= `'vertical'` | `'horizontal'`
lane-marking-type ::= `'dashed'` | `'solid'`

action-source ::= `'layer-M'` | `'layer-P'` | `'layer-R-rule-'` IDENTIFIER
agent-action ::=
  `'change-speed'` | `'change-orientation'` | `'signal-left'`
  `'signal-right'` | `'flash-headlights'` | `'honk-horn'`
  `'hazard-lights'` | `'fog-lights'`
action-arguments ::= PROLOG LIST STRUCTURE
intention-arguments ::= PROLOG LIST STRUCTURE | `'_'`

Bibliography

[AC87] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 268–272, 1987.

[AD90] Tracy L. Anderson and Max Donath. Animal behaviour as a paradigm for developing robot autonomy. *Robotics and Autonomous Systems*, 6(1&2):145–168, 1990.

[AIS90] Jose A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 735–740. Morgan Kaufmann: San Mateo, CA, 1990.

[APW90] Panos J. Antsaklis, Kevin M. Passino, and S.J. Wang. An introduction to autonomous control systems. In *Proceedings Fifth IEEE International Symposium on Intelligent Control*, Philadelphia, PA, 1990.

[Asi50] Isaac Asimov. *I, Robot*. Doubleday: New York, NY, 1950.

[BD90] John Bresina and Mark Drummond. Integrating planning and reaction – a preliminary report. In James Hendler, editor, *Proceedings AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*, Systems Research Center, University of Maryland, MD, 1990.

[BG88] Alan H. Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann: Palo Alto, CA, 1988.

[BIP88] Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(4):349–355, November 1988.

[Bro86] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[Bro91a] Rodney A. Brooks. Intelligence without reason. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 569–595, 1991.

[Bro91b] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[Car90a] Sandra Carberry. Incorporating default inferences into plan recognition. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 471–478, 1990.

[Car90b] Sandra Carberry. *Plan Recognition in Natural Language Dialogue*. MIT Press: Cambridge, MA, 1990.

[CEG+91] Jaime Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Craig Knoblock, Steve Minton, and Manuela Veloso. Prodigy: An integrated architecture for planning and learning. *ACM SIGART Bulletin*, 2(4):51–55, 1991.

[CGHH89] Paul R. Cohen, Michael L. Greenberg, David M. Hart, and Adele E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, 1989.

[Cha90] David Chapman. Planning for conjunctive goals. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 537–558. Morgan Kaufmann: San Mateo, CA, 1990.

[CL87] Philip R. Cohen and Hector J. Levesque. Persistence, intention, and commitment. In M.P. Georgeff and A.L. Lansky, editors, *Reasoning about Actions and Plans - Proceedings 1986 Workshop*, pages 297–340. Morgan Kaufmann: Los Altos, CA, 1987.

[CL91] Arie A. Covrigaru and Robert K. Lindsay. Deterministic autonomous agents. *AI Magazine*, 12(3):110–117, 1991.

[CM81] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag: Berlin, Germany, 1981.

[Coh91] Paul R. Cohen. A survey of the Eighth National Conference on Artificial Intelligence: Pulling together or pulling apart. *AI Magazine*, 12(1):16–41, 1991.

[Dav90] Ernest Davis. *Representations of Commonsense Knowledge*. Morgan Kaufmann: San Mateo, CA, 1990.

[Daw76] Richard Dawkins. *The Selfish Gene*. Oxford University Press: Oxford, UK, 1976.

[DB86] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 49–54, 1986.

[DH88] Randall Davis and Walter Hamscher. Model-based reasoning: Troubleshooting. In Howard E. Shrobe and AAAI, editors, *Exploring Artificial Intelligence*, pages 297–346. Morgan Kaufmann: San Mateo, CA, 1988.

[DK90] Mark E. Drummond and Leslie Pack Kaelbling. Integrated agent architectures: Benchmark tasks and evaluation metrics. In *Proceedings DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 408–411. Morgan Kaufmann: San Mateo, CA, 1990.

[dKW86] Johan de Kleer and Brian C. Williams. Reasoning about multiple faults. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 132–139, 1986.

[DM89] Edmund H. Durfee and Thomas A. Montgomery. MICE: A flexible testbed for intelligent coordination experiments. In *Proceedings Ninth Workshop on Distributed Artificial Intelligence*, Rosario Resort, Eastsound, WA, 1989.

[DM90] Edmund H. Durfee and Thomas A. Montgomery. A hierarchical protocol for coordinating multiagent behaviours. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 86–93, 1990.

[DR91] Joseph Downs and Han Reichgelt. Integrating classical and reactive planning within an architecture for autonomous agents. In *Proceedings First European Workshop on Planning*, Birlinghoven, Germany, 1991.

[DW90] Jon Doyle and Michael P. Wellman. Rational distributed reason maintenance for planning and replanning of large-scale activities (preliminary report). In *Proceedings DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 28–36. Morgan Kaufmann: San Mateo, CA, 1990.

[DW91] Thomas L. Dean and Michael P. Wellman. *Planning and Control*. Morgan Kaufmann: San Mateo, CA, 1991.

[Fir87] James R. Firby. An investigation into reactive planning in complex domains. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 202–206, 1987.

[FN90] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 88–97. Morgan Kaufmann: San Mateo, CA, 1990.

[Fod83] Jerry A. Fodor. *The Modularity of Mind*. MIT Press: Cambridge, MA, 1983.

[Gal90] Julia R. Galliers. The positive role of conflict in cooperative multi-agent systems. In Jean-Pierre Müller and Yves Demazeau, editors, *Decentralized A.I.* Elsevier (North-Holland): Amsterdam, NL, 1990.

[Gas91] Les Gasser. Social conceptions of knowledge and action: DAI foundations and open systems semantics. *Artificial Intelligence*, 47:107–138, 1991.

[GBH87] Les Gasser, Carl Braganza, and Nava Herman. MACE: A flexible testbed for distributed AI research. In Michael N. Huhns, editor, *Distributed Artificial Intelligence*, pages 119–152. Morgan Kaufmann: Los Altos, CA, 1987.

[Geo83] Michael P. Georgeff. Communication and interaction in multi-agent planning. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 125–129, 1983.

[Geo90] Michael P. Georgeff. Planning. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 5–25. Morgan Kaufmann: San Mateo, CA, 1990.

[GI89] Michael P. Georgeff and François Felix Ingrand. Decision-making in embedded reasoning systems. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 972–978, 1989.

[GLS87] Michael P. Georgeff, Amy L. Lansky, and Marcel J. Schoppers. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Technical Note 380, SRI International, Menlo Park, CA, April 1987.

[GN87] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann: Los Altos, CA, 1987.

[Gre87] Richard L. Gregory, editor. *The Oxford Companion to the Mind*. Oxford University Press: Oxford, UK, 1987.

[HC90] Adele E. Howe and Paul R. Cohen. Responding to environmental change. In *Proceedings DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 85–92. Morgan Kaufmann: San Mateo, CA, 1990.

[HF90] Steve Hanks and R. James Firby. Issues and architectures for planning and execution. In *Proceedings DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 59–70. Morgan Kaufmann: San Mateo, CA, 1990.

[HH88] Bernardo A. Huberman and Tad Hogg. The behaviour of computational ecologies. In Bernardo A. Huberman, editor, *The Ecology of Computation*, pages 77–115. Elsevier (North Holland): Amsterdam, NL, 1988.

[HHC90] Adele E. Howe, David M. Hart, and Paul R. Cohen. Addressing real-time constraints in the design of autonomous agents. COINS Technical Report 90-06, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, 1990.

[Hof79] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Penguin Books: London, UK, 1979.

[HR88] Barbara Hayes-Roth. Making intelligent systems adaptive. Report STAN-CS-88-1226, Stanford University, Stanford CA, October 1988.

[HR90] Barbara Hayes-Roth. Architectural foundations for real-time performance in intelligent agents. *The Journal of Real-Time Systems*, 2:99–125, 1990.

[HTD90] James Hendler, Austin Tate, and Mark Drummond. AI planning: Systems and techniques. *AI Magazine*, 11(2):61–77, 1990.

[Kae87] Leslie Pack Kaelbling. An architecture for intelligent reactive systems. In M.P. Georgeff and A.L. Lansky, editors, *Reasoning about Actions and Plans - Proceedings 1986 Workshop*, pages 395–410. Morgan Kaufmann: Los Altos, CA, 1987.

[Kae91] Leslie Pack Kaelbling. Specifying complex behaviours for computer agents. In Luc Steels and Barbara Smith, editors, *Proceedings Eighth Conference on Artificial Intelligence and the Simulation of Behaviour*. Springer-Verlag: London, UK, 1991.

[KG91] David N. Kinny and Michael P. Georgeff. Commitment and effectiveness of situated agents. Technical Note 17, Australian Artificial Intelligence Institute, Carlton 3053, Australia, April 1991.

[Kim90] Ted Kim. XWIP Reference Manual Version 0.5. Technical report, University of California, Los Angeles, CA, 1990.

[Kir91] David Kirsh. Today the earwig, tomorrow man? *Artificial Intelligence*, 47:161–184, 1991.

[Kon83] Kurt Konolige. A deductive model of belief. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 377–381, 1983.

[KR92] George Kiss and Han Reichgelt. Towards a semantics of desires. In Eric Werner and Yves Demazeau, editors, *Decentralized A.I. 3*. Elsevier (North-Holland): Amsterdam, NL, 1992.

[Lai91] John E. Laird, editor. *Special Section on Integrated Cognitive Architectures*. ACM SIGART Bulletin, 2(4):12–184, 1991.

[Lan88] Pat Langley. Machine learning as an experimental science. *Machine Learning*, 3:5–8, 1988.

[Lat91] Jean Claude Latombe. A fast path planner for a car-like indoor mobile robot. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 659–665, 1991.

[LC83] Victor R. Lesser and Daniel D. Corkill. The distributed vehicle monitoring testbed: A tool for investigating distribution. *AI Magazine*, 4(3):15–33, 1983.

[LCN90] Hector J. Levesque, Philip R. Cohen, and José H.T. Nunes. On acting together. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 92–99, 1990.

[LD90] Pat Langley and Mark Drummond. Toward an experimental science of planning. In *Proceedings DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 109–114. Morgan Kaufmann: San Mateo, CA, 1990.

[Lee89] Mark H. Lee. *Intelligent Robotics*. Open University Press: Milton Keynes, UK, 1989.

[Len75] Douglas B. Lenat. BEINGS: Knowledge as interacting experts. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 126–133, 1975.

[LR90] John E. Laird and Paul S. Rosenbloom. Integrating execution, planning, and learning in SOAR for external environments. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 1022–1029, 1990.

[Mae90] Pattie Maes. Situated agents can have goals. *Robotics and Autonomous Systems*, 6(1&2):49–70, 1990.

[McD90] Drew McDermott. Planning and acting. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 225–244. Morgan Kaufmann: San Mateo, CA, 1990.

[MD88] Mark S. Miller and K. Eric Drexler. Markets and computation: Agoric open systems. In Bernardo A. Huberman, editor, *The Ecology of Computation*, pages 133–176. Elsevier (North Holland): Amsterdam, NL, 1988.

[Mil29] A.A. Milne. *Winnie Ille Pu*. Methuen: London, UK, 1929. Translated by Alexander Lenard.

[Min86] Marvin L. Minsky. *The Society of Mind*. Simon and Schuster: New York, NY, 1986.

[Mit90] Tom T. Mitchell. Becoming increasingly reactive. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 1051–1058, 1990.

[Mor88] Hans Moravec. *Mind Children - The Future of Robot and Human Intelligence*. Harvard University Press: Cambridge, MA, 1988.

[MRS82] Gordon I. McCalla, Larry Reid, and Peter F. Schneider. Plan creation, plan execution and knowledge acquisition in a dynamic microworld. *International Journal of Man-Machine Studies*, 16:89–112, 1982.

[New82] Allen Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

[Oga91] Gary H. Ogasawara. A distributed, decision-theoretic control system for a mobile robot. *ACM SIGART Bulletin*, 2(4):140–145, 1991.

[PGA86] David L. Poole, Randy G. Goebel, and Romas Aleliunas. Theorist: A logical reasoning system for defaults and diagnosis. Research Report CS-86-06, University of Waterloo, Waterloo, Ont., February 1986.

[Poo88] David Poole. Representing knowledge for logic-based diagnosis. In *Proceedings International Conference on Fifth Generation Computer Systems*, pages 1282–1289, 1988.

[PR90] Martha E. Pollack and Marc Ringuette. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 183–189, 1990.

[RG85] Jeffery S. Rosenschein and Michael R. Genesereth. Deals among rational agents. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 91–99, 1985.

[RK91] Elaine Rich and Kevin Knight. *Artificial Intelligence.* McGraw-Hill: New York, NY, 2nd edition, 1991.

[Rus89] Stuart J. Russell. Execution architectures and compilation. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 15–20, 1989.

[RW89] Stuart Russell and Eric Wefald. Principles of metareasoning. In *Proceedings International Conference on Principles of Knowledge Representation and Reasoning*, pages 400–411, 1989.

[Sac90] Earl D. Sacerdoti. The nonlinear nature of plans. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 162–170. Morgan Kaufmann: San Mateo, CA, 1990.

[SC85] Michael Sullivan and Paul R. Cohen. An endorsement-based plan recognition program. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 475–479, 1985.

[Sch87] M.J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

[SH88] J. Sanborn and J. Hendler. A model of reaction for planning in dynamic environments. *International Journal of Artificial Intelligence in Engineering*, 3(2):95–102, 1988.

[SH90] Lee Spector and James Hendler. Knowledge strata: Reactive planning with a multi-level architecture. Technical Report CS-TR-2564, Computer Science Dept., University of Maryland, College Park, MD, 1990.

[Sho90] Yoav Shoham. Agent-Oriented Programming. Technical Report STAN-CS-90-1335, Computer Science Dept., Stanford University, Palo Alto, CA, 1990.

[Sim81] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press: Cambridge, MA, 2nd edition, 1981.

[SM90] Yoav Shoham and Drew McDermott. Problems in formal temporal reasoning. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 581–587. Morgan Kaufmann: San Mateo, CA, 1990.

[SR88] John A. Stankovic and Krithi Ramamrithan, editors. *Hard Real-Time Systems*, chapter 1: Introduction, pages 1–11. Computer Society of the IEEE: Washington, DC, 1988.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press: Cambridge, MA, 1986.

[TMC81] Perry W. Thorndyke, Dave McArthur, and Stephanie Cammarata. AUTOPILOT: A distributed planner for air fleet control. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 171–177, 1981.

[Tur50] A. M. Turing. Computing machinery and intelligence. *MIND*, LIX(236):433–460, 1950.

[VB90] Steven Vere and Timothy Bickmore. A basic agent. *Computational Intelligence*, 6(1):41–60, 1990.

[WB87] Yorick Wilks and Afzal Ballim. Multiple agents and the heuristic ascription of belief. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 118–124, 1987.

[Wil85] David E. Wilkins. Recovering from execution errors in SIPE. *Computational Intelligence*, 1:33–45, November 1985.

[Wil86] David E. Wilkins. High-level planning in a mobile robot domain. Technical Note 388, SRI International, Menlo Park, CA, 1986.

[Wil90] Stewart W. Wilson. The animat path to AI. In J.A. Meyer and S.W. Wilson, editors, *From Animals to Animats: Proceedings of the First International Conference on the Simulation of Adaptive Behaviour*. MIT Press: Cambridge, MA, 1990.

[Woo90] Sharon Wood. *Planning in a Rapidly Changing Environment*. PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK, 1990.