A BELIEF-DESIRE-INTENTION ARCHITECTURE WITH A LOGIC-BASED PLANNER FOR AGENTS IN STOCHASTIC DOMAINS

by

GAVIN B. RENS

submitted in accordance with the requirements for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR : Prof. E. van der Poel
JOINT SUPERVISOR : Dr. A. Ferrein

February 2010

SUMMARY

This dissertation investigates high-level decision making for agents that are both goal and utility driven. We develop a partially observable Markov decision process (POMDP) planner which is an extension of an agent programming language called DTGolog, itself an extension of the Golog language. Golog is based on a logic for reasoning about action—the *situation calculus*. A POMDP planner on its own cannot cope well with dynamically changing environments and complicated goals. This is exactly a strength of the belief-desire-intention (BDI) model: BDI theory has been developed to design agents that can select goals intelligently, dynamically abandon and adopt new goals, and yet *commit* to intentions for achieving goals. The contribution of this research is twofold: (1) developing a *relational* POMDP planner for cognitive robotics, (2) specifying a preliminary BDI architecture that can deal with stochasticity in action and perception, by employing the planner.

Key terms

cognitive robotics, intelligent agents, planning, partial observability, POMDP, belief-desire-intention paradigm, BDI theory, architecture, high-level control, logic, situation calculus, Golog

Contents

List of Figures

List of Algorithms

List of Tables

For my wife and parents.

Acknowledgements

Esmarie, my wife, has encouraged and supported me all the way. I am sincerely grateful to her. Tommie Meyer and Arina Britz opened doors for me and provided ongoing general support once through the doors. Alexander Ferrein and Etienne van der Poel gave guidance and suggestions concerning theoretical aspects of the work. Etienne, thank you for taking care of the administrative stuff on the UNISA side. Thanks to Gerhard Lakemeyer, Tommie Meyer and Alexander Ferrein for arranging for me to visit the Knowledge-Based Systems Group at the Technical University of Aachen (RWTH) for four months in 2008; I want to especially thank Alexander for his personal support during my stay in Aachen. Lastly, the completion of this dissertation would have been virtually impossible without the two-year studentship for full-time study awarded by the Meraka Institute of the CSIR, South Africa.

Chapter 1

Introduction

1.1 Automated Decision-making in the Real World

Some systems have a range of options available to them, and depending on the options chosen, the output changes; some outputs may be more desirable and others less so. A robot (or software agent) may be viewed as a system with available options. The actions the robot can perform are the system parameters that can be tuned, and the system's output is called the *behavior* in cognitive robotics and agent systems. Autonomous robots—the kind we are interested in—are automated systems. For an autonomous robot to behave as expected, at the standard expected by the robot's designers, the robot must decide on appropriate actions to achieve the expected standard and kind of behavior. This is *automated decision-making*.

As we move into the future, robots will be expected to operate effectively and reliably in the *real world*, that is, in the world we humans live in. The world wide web is also becoming extremely complex and intelligent autonomous agents are being designed and deployed to gather information or do other tasks (for example, network security) on our behalf. These *softbots* 'living' in cyber space will thus also benefit from methods in "automated decision-making in the real world".

For robots and agents to be effective in complex and changing environments with many so-called 'grey-areas', many approaches and methods have and are being developed in cognitive robotics and intelligent agent systems. With this work, we hope to contribute to making agents more intelligent, autonomous and robust.

1.2 Motivation for this Dissertation

Broadly speaking, there are two kinds of driving forces or 'motivation' for knowledge-based agents: goals and utility. Goal driven agents have a specific state they want to reach and there is essentially no value for being in other states. Utility driven agents though, have tasks that can be achieved continually and incrementally; rewards are scattered over many states and they must be collected. There is typically no specific goal state to reach in domains where utility driven agents live.

This dissertation investigates high-level decision making where objectives are formulated as goals with utility attached to the achievement of the goals. In other words, the agents researched here are both goal and utility driven.

Traditionally, plan-based agents that include generative planning (as opposed to utilizing pre-compiled plans) would generate a complete plan to reach a *specific fixed* goal, then execute the plan. If plan execution monitoring is available, the agent would replan from scratch when the plan becomes invalid. Due to the time requirements for generating complete plans, the plan may be invalid by the time it is executed. This is because the world may change substantially during plan generation. Therefore, belief-desire-intention (BDI) architectures take a different approach.

BDI theory is based on the philosophy of practical reasoning [15]. It offers flexibility in planning beyond traditional planning for agents, by reasoning over *different goals*. That is, an agent based on BDI theory can adapt to changing situations by focusing on the pursuit of the most appropriate goal at the time. Typically, an appropriate plan to achieve an adopted goal is selected from a data base of plans. Although a plan that satisfies certain constraints (for example, does not conflict with other adopted plans, is executable, etcetera) will be adopted, it may not be the most appropriate plan in existence. A plan that is generated with the agent's current knowledge for guidance, may be more appropriate. BDI agents can also make rational decisions as to when to replan if a plan becomes invalid, reducing the amount of replanning, thus increasing the agent's reactivity. Note that the BDI paradigm is, however, not the only approach to replanning (see for example, Likhachev *et al.* [65]).

In general, BDI architectures do not make use of plan generation, they rather draw on plan libraries. While with BDI approaches, an agent can reason over several goals, the agent lacks some flexibility by not being able to generate suitable plans on demand. Therefore, in this dissertation, we aim at integrating a partially observable Markov decision process (POMDP) planner into a BDI architecture to combine benefits of the architecture with the ability to *generate* plans. Moreover, we want to supply models that are as realistic as possible; we therefore decided on employing POMDPs.

We design the POMDP planner as a new dialect of the agent programming language Golog [63]. Golog has its basis in a logic for reasoning about actions, called the *situation calculus* [66]. An advantage of using a Golog implementation for the planner is that the integration of beliefs into the situation calculus has previously been done [2] and this work can be used for formulating POMDPs. Further, given a background action theory, an initial state and a goal state (or 'reward function' in POMDPs), Golog programs essentially constrain and specify the search space of available actions. The resulting plan (*policy* in POMDP terms) is a Golog program which can be executed directly by an agent.

1.3 Assumptions and Delimitations of this Dissertation

We shall refer to 'robot' and 'agent' interchangeably; both terms refer to autonomous intelligent embedded systems. See Appendix A for definitions of *robot* and *agent*.

The work reported on in this dissertation concerns a certain class of robot: those robots that consult a database of facts and rules (called a *knowledge-base* (KB)) and that execute *plans*. It is conceivable that a knowledge-based robot needs not be a plan-based robot, for instance, a robot may react only according to rules stored in a KB, 'triggered' due to sensory inputs. It is also conceivable that a plan-based robot needs not be a knowledge-based robot, in the sense that the robot executes some *plan* selected according to its sensory inputs, but the robot does not have separate rules.

An agent may be both knowledge- and plan-based in at least two senses: (1) The facts and rules in its KB are used in a reactive manner and/or its facts and rules are used to select plans. (2) A plan is generated in consultation with its KB and current sensor inputs, and that plan is executed. The agents we shall be most concerned with in the dissertation are knowledge- and plan-based in sense 2, however, to establish a context, agents of sense 1 will also be discussed.

In robotics, there is a rough distinction between *high-level* and *low-level* computation. High-level computation may be likened to conscious reasoning, that is planning, decision-making and judgment, whereas low-level computation may be likened to subconscious processes such as scene and object identification, posture maintenance and stimulus-response behavior. For robots in the real world to be useful, they are (arguably) required to perform both high- and low-level computation. In the present work, we abstract away from low-level computation and either simulate its presence or simplify the problem enough so as to make low-level computation a non-issue. In other words, the present work focuses on the high-level reasoning of robots and agents.

We would like to focus the reader's attention early on, on an issue of potential confusion. When creating complex systems such as robots, a robot can be described or represented at several

levels of abstraction, from the physical mechanics and electronics to behaviors, intentions and knowledge. One can distinguish between three levels of abstraction:

1. a *paradigm* is an approach, framework or perspective;
2. a *theoretical implementation* provides detail to a paradigm by producing an algorithm or a formal specification for a whole architecture or system or part thereof;
3. a *practical implementation* is a physical, usable system; an end product—it implements a theoretical implementation such that it has a practical benefit or can be observed in operation.

In this dissertation, *paradigms* describe structures at a more abstract level than *architectures*. And a *practical implementation* is more concrete than a *theoretical implementation*.

Note though, that the relationship of these terms as set out above, are not conventional. For example, broadly speaking, 3T, PRS, UM-PRS and JAM are all systems for designing and controlling synthetic agents. 3T, an instance of the three-layer paradigm[1], is called an "architecture" by its designers [8, 36], while the paradigm itself is also referred to as an architecture. Also, PRS is referred to by its creators [40] as a "system", whereas Wooldridge and Jennings [115] refer to PRS as an "architecture". Lee *et al.* [60] *implement* PRS with the C++ language, resulting in another "system", called UM-PRS. Huber [44] talks about JAM as an "architecture" which draws upon PRS. In the literature, whether a paradigm or theoretical implementation is discussed (in terms of our three levels of abstraction), it is usually referred to as an architecture. In the literature, the term *model* is frequently used instead of *paradigm*. This convention will be adopted here from time to time. But when making a deeper study of agents, from theory to practice, as is done here, one needs to *distinguish* between levels of abstraction.

The theoretical aspects of this dissertation support the union of POMDP planning and BDI deliberation as a hybrid architecture, and the experiment results support the idea as a proof of concept. However, we shall also argue that there is scope for improvement of the hybrid architecture. A more formal specification and analysis of the new architecture is still needed. Optimal POMDP solvers are known to be highly inefficient, thus, some optimizations to the solver we developed are also investigated. Nevertheless, more work is required in that area too.

Parts of the presented material have already been published [85, 86].

During the preparation of this dissertation, the assumption was made that the readers will have, at least, a rudimentary knowledge of set theory, basic algebra, and first-order predicate logic. Alternatively, an under-graduate level qualification in computer science or a similar discipline is assumed. Readers with this basic background can acquire the further necessary background specific to this dissertation, by referring to Appendix A and Chapter 2, in any order.

[1] The three-layer paradigm is discussed in Appendix A.

Notation: Throughout the document, $\doteq$ and $\stackrel{def}{=}$ denote 'is defined as'. $|x|$ is the absolute value of x. Italicized words should be read with emphasis and have special significance. Words in single quotation marks, for example 'X', should be read: *so called X*, or: *X in a manner of speaking*. Words, phrases, etcetera, in double quotation marks, for example "X", mean that X has been mentioned before, in this or other documents.

In conclusion of this section and in preparation for the next section, an analysis of the dissertation's title follows: "A Belief-Desire-Intention Architecture with a Logic-based Planner for Agents in Stochastic Domains".

- "A Belief-Desire-Intention Architecture": This is a specific kind of architecture for high-level reasoning in agent systems;
- "with a Logic-based Planner": The architecture of interest has a logic-based planner, that is, a planner based on formal logic;
- "for Agents": The architecture and planner are for the automated control of a specific kind of system: an agent system;
- "in Stochastic Domains": The agents to be controlled inhabit the real world or real-world-like domains, that is, stochastic domains.

1.4 Goals and Contributions of this Dissertation

Broadly, the goal *and* contribution of the dissertation is to develop a new logic-based planner for stochastic domains and to modify the basic belief-desire-intention architecture to employ this planner for use by agents and robots living in stochastic domains.

1.4.1 Thesis Statement

It is possible to define a belief-desire-intention architecture that employs a logic based planner that generates control policies for agents inhabiting partially observable stochastic domains, such that the performance of these agents is reasonable or such that the hybrid architecture shows *clear potential* for controlling agents, producing reasonable performance.

1.4.2 Research Questions

1. Can an existing language, DTGolog, be extended to generate policies for partially observable Markov decision process (POMDP) problems?

2. Can a logic-based POMDP planner be integrated with the belief-desire-intention architecture? That is, is it possible to specify a 'reasonable' hybrid BDI/POMDP-planner architecture?

3. Is there a performance gain in an agent when the agent is controlled by policies generated from the logic-based POMDP planner as compared to being controlled by policies generated from the logic-based planner that assumes full observability?

4. Is there a performance gain in an agent when the agent is controlled by the hybrid BDI/POMDP-planner architecture as compared to being controlled by a simpler architecture that employs the new POMDP planner?

1.5 Outline

The next chapter covers some necessary background theory; it covers decision theory and the theory on which BDI architectures are based. The reader may read this background chapter before Chapter 3 or consult it in conjunction. Chapter 3 presents a literature survey of related work. The literature covered is divided into three categories: (i) logics for dynamical stochastic domains, (ii) Golog dialects for stochastic domains and (iii) BDI-based architectures with generative planning. Chapter 4 describes the new logic-based POMDP planner by way of an extension to an existing logic-based planner. We suggest a hybrid architecture in Chapter 5 that integrates the logic-based POMDP planner with the BDI architecture. Due to the inherent intractability of finding solutions to POMDP problems, we need to look at ways to optimize the new planner. This is addressed in Chapter 6. Then, in Chapter 7, we present results of experiments to assess the new planner and the new architecture from various angles. The last chapter gives a summary of this work, conclusions are drawn, some final issues are discussed and directions for research that could flow from the current research are mentioned.

The first appendix is a useful reference to models for high-level control of agents and robots, introducing the common concepts and formalisms in the field. The second appendix provides source code of most of the implementations used for this research. The last two appendices are published papers relevant to the present work.

Chapter 2

Formal Background

DTGolog is a dialect of the agent programming language Golog that has been extended to deal with (fully observable) decision theoretic (DT) domains. In this work, we extend DTGolog to PODTGolog (partially observable decision theoretic Golog). To understand DTGolog and its extension PODTGolog, it is thus necessary to formally introduce decision theory and Golog.

This chapter is in three parts: The first part (Section 2.1) covers what the reader needs to know concerning decision theory; the fully observable Markov decision process (FOMDP) is covered first and then the partially observable Markov decision process (POMDP). The second part introduces DTGolog systematically, beginning with the introducing of the formal logic called the *situation calculus* in Section 2.2.1. Golog, presented in Section 2.2.2, is a programming language based on the situation calculus. How Golog is extended to DTGolog is explained in Section 2.2.3. The third part of this chapter (Section 2.3) introduces belief-desire-intention theory, the foundation theory of the architecture developed in this dissertation.

2.1 Decision Theory

Suppose an agent can decide which state of the world it prefers by looking at certain attributes of the world, somehow combining the values of these attributes and coming up with a single real number reflecting its preference for the state the world is in. The agent could then consider different possible worlds, and decide which of these it prefers being in. In utility theory, the attribute values are called utilities. Let X_A be the vector of utilities the agent perceives when in state A. The function that the agent uses to combine the utilities is called the agent's utility function $U(X)$, where X is a vector of utilities and U returns a real number. Utility theory says that an agent can and does attach utilities to attributes of its world, and that the agent has a preference for one state of the world over another, based on its utilities [88, 20, 93, 107].

Imagine a scenario where $U(X_A) \gg U(X_B)$ for some agent considering states A and B. If the

agent knows that it will be in state A with probability 0.9 and in state B with probability 0.1 after doing the action GO_1, and that it will be in state A and B with probability 0.5 each, after doing the action GO_2—then rationally, the agent should choose to GO_1. This simple example illustrates that agents should consider the *probability* of being in a state in combination with the utility of the state when making decisions.

"Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called **decision theory**:

Decision theory = probability theory + utility theory," [88, p. 465].

A person, robot or an agent in general may intend to step forward, that is, perform a 'forward-step' action, but find that it slipped left (on a wet tiled floor, for example). This 'forward-step' action is thus *nondeterministic* and the 'slip-left' action is one *outcome* of 'forward-step'. Let GO represent any one of an agent's available actions. Suppose GO_i is a nondeterministic outcome of performing GO, with i ranging over all possible outcomes for GO and $Pr(GO_i \mid E)$ the probability that GO will turn out as GO_i, given E, where E summarizes the agent's evidence about the world. Decision theory claims that a rational agent should make the decision that will maximize its expected utility (the MEU assumption) [88, 20], that is, Equation 2.1 is maximized by choosing the action GO that results in the highest value:

$$EU(GO \mid E) = \sum_i Pr(GO_i \mid E) \times U(X_i), \tag{2.1}$$

where X_i is the utility vector of the state resulting from performing GO_i in the current state.

2.1.1 Fully Observable Markov Decision Processes

Most of the theory in this subsection can be found in Russell and Norvig [88], Kaelbling *et al.* [48] and Boutilier et al. [11].

It has become convention to interpret 'Markov decision process' (MDP) as referring to the fully observable Markov decision process (FOMDP). (FO)MDPs assume full observability of the applicable system or environments. That means, after any action is executed, the resulting state is deterministically/completely known. Furthermore, a Markov process makes the Markov assumption: that the next state can be completely determined by knowledge of only the current state and the action executed. That is, the next state is independent of any states that the system was in before the current state.

An MDP is a *decision* process and thus facilitates making decisions as to which actions to take, given its previous actions. That is, an MDP facilitates choosing actions in a dynamical system.

Lastly, in an MDP, utilities must be calculated *additively* [88]. This means the utility function is essentially linear in the attributes.

Given these constraints, the class of Markov decision processes is large. The way in which the MDP model is defined in the next section excludes otherwise feasible models. In other words, the definition presented, selects the subclass applicable to the present research.

The Model

An MDP model is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, s^0 \rangle$ such that

- $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is a finite set of states of the world (that the agent can be in);
- $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$ is a finite set of actions; actions include those that the agent can choose to execute and those that are the nondeterministic outcomes of the chosen action;
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \Pi$ is the *state-transition function*, a probability distribution Π over all (world state, agent action, world state) triples (we write $T(s, a, s')$ to mean the probability of being in s' after performing action a in state s);
- $\mathcal{R} : \mathcal{S} \to \mathbb{R}$ is the *reward function*, giving the expected immediate reward gained by the agent, for any world state and agent action (we write $\mathcal{R}(s)$ to mean the reward gained for being in state s)[1];
- s^0 is the initial state of the agent.

A state at step t is denoted s^t and is defined by state features. Each feature can have a range of values and each $s_i \in \mathcal{S}$ is a (unique) 'snapshot' of the state features and their assigned values. Initially, $t = 0$.

To clarify the kinds of actions that make up $\mathcal{A}$, consider this: at one time, a robot may choose to execute a_1, but the environment causes the action to be realized as a_2; to an outsider, it would seem as though the robot chose to do a_2. At another time, the robot may intend to execute a_2, but the environment 'chooses' a_1 as the resulting outcome. Executing, say, a_1 may of course have a_1 as an actual result.

$\mathcal{T}$ represents how the agent's actions change the world—more accurately, the likelihood that the world will be in a certain new state, given a certain action is executed while the world is in a certain current state. $\mathcal{T}$ is also called the *transition model* of the domain, because it models system transitions. This work does not consider how to learn or calculate transition models; the decision theoretic agents we consider here are supplied with a transition model.

[1] $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is another definition for reward functions within the MDP model, however, our definition of $\mathcal{R}$ suits our application better, and it does not affect the problems or their solutions fundamentally [88].

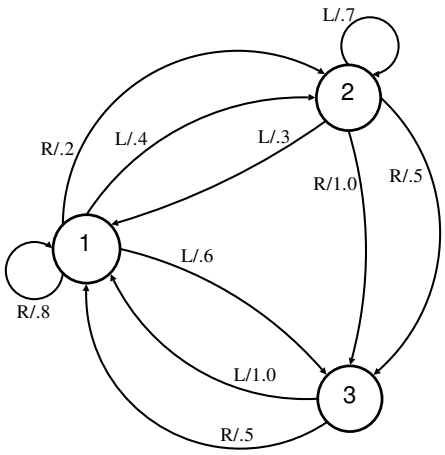

Figure 2.1: A state-transition diagram for two actions: L (move left) and R (move right).

If state transitions are modeled differently depending on the number of steps taken, the system is *nonstationary*. If the agent will have the same transition behavior from state to state, regardless of the number of actions performed by the agent, then we say the agent is in a *stationary* system. Only agents modeled in stationary systems are considered in this work.

The formalism used to represent an agent's action and possible states is the *state-transition diagram* [11] (also known as an *influence diagram* [20]). Figure 2.1 is an example state-transition diagram with three states (circles) and two possible actions. An arc labeled a/p leaving state s and entering s' means that when a is executed in s, it is realized with probability p as the action outcome that puts the agent in state s'. "A stationary Markov process can also be represented using a [weighted] *state-transition diagram*" [11, p. 8].

The reward function $\mathcal{R}$ is essentially a kind of utility function.

Determining a Policy

Just as a single action can be chosen to maximize the utility expected in the next state, a *sequence* of actions (of a chosen length N) can be chosen that will maximize the expected utility after a sequence of N actions. The *optimality prescription* of utility theory states: Maximize "the expected sum of rewards that [an agent] gets on the next k steps," [48].

A *policy* π is a conditional plan that tells a robot what action to perform when in any state.[2] For each $s \in \mathcal{S}$, $\pi(s)$ is the recommended action for an agent in s,

$$\pi : \mathcal{S} \rightarrow \mathcal{A},$$

[2]Technically, a *plan* is executed by an agent from a specific initial state and it may simply be a sequence of actions that the agent is expected to follow blindly. A plan may or may not be conditioned on observations. A *policy* on the other hand, instructs an agent which action to take given any (predefined) state the agent may be in and given the last observation. As such, a policy describes the behavior of an agent that employs the policy, but one cannot say that the behavior of an agent is fully determined by the *plan* it is executing.

and because only stationary policies are considered, $\pi(s)$ always recommends the same action, regardless of how the agent got there. A policy is necessary because a robot may never know for sure in what state it will end up after it executes a (nondeterministic) action: due to nondeterminism, action recommendation is conditional on current state, not on action history. Given the same initial state and policy, a robot's action history (environment history) may well be different for different runs of N actions, due to the stochasticity in the environment. "The quality of a policy is therefore measured by the *expected* utility of the possible environment histories generated by that policy," [88, p. 615].

The *horizon* of a process is the number of steps that will be considered before the total utility collected by the robot or agent is assessed. For example, a process with horizon 9 will require 9 actions/steps to be taken before the total utility of the system is determined; before 9 steps, the system cannot be in a desired state. A 9-horizon system will have an optimal policy if the expected utilities collected is a maximum after exactly 9 steps from the initial state s^0. A *finite* horizon system is one that is not infinite[3]—it considers a constant number of steps. The work in this dissertation employs only finite horizons. The reason for this will become clear later in this chapter.

The *value* $V_{\pi,h}(s)$ of a state s with respect to a specific policy π is the expected sum of rewards for h steps[4] and is defined by Equation 2.2:

$$V_{\pi,h}(s) = E\left[\sum_{t=0}^{h} \mathcal{R}(s^t) \mid \pi, s^0 = s\right]. \tag{2.2}$$

Clearly $V_{\pi,0}(s) = \mathcal{R}(s)$ and $V_{\pi,1}(s) = \mathcal{R}(s) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi(s), s')\mathcal{R}(s') = \mathcal{R}(s) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi(s), s')V_{\pi,0}(s')$. In general, the *value function* is

$$V_{\pi,h}(s) = \mathcal{R}(s) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi(s), s')V_{\pi,h-1}(s'). \tag{2.3}$$

There are various optimality criteria for policies. One criterion is that, *on average, for any* initial state in $\mathcal{S}$, the policy will yield the maximum possible expected rewards. In the present work, we focus on the criterion that the optimal policy (denoted π^*) for a finite horizon system (of horizon h) is the policy that results in the maximum possible value *for a specified* initial state. An optimal policy π^* thus satisfies

$$\pi^*(s^t) = \arg\max_a \left(\mathcal{R}(s^t) + \sum_{s^{t+1} \in \mathcal{S}} \mathcal{T}(s^t, a, s^{t+1})V_{t-1}(s^{t+1})\right), \text{ for all } 0 \leq t < h. \tag{2.4}$$

$V_h(s)$ is the value function that is independent of a policy. Note that h implies that π^* must recommend h actions; $0 \leq t < h$ ensures this.

[3] Infinite horizon systems have some advantages over finite horizon systems, but cannot always be used.

[4] Equation 2.2 can also be read as the value of the *policy* π—followed for h steps, starting at s.

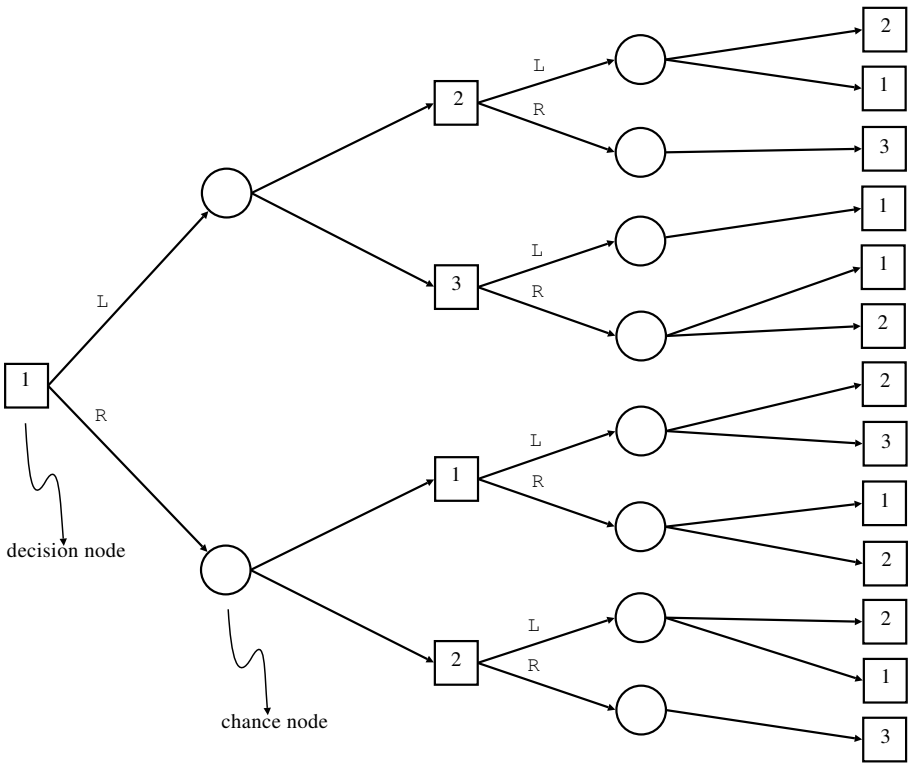

Figure 2.2: A decision tree for deciding between two nondeterministic actions.

For some applications—in particular, the DTGolog planner [13] and its extension developed in this dissertation—we do not require policies for an infinite horizon. One advantage of calculating finite horizon policies, is that $\pi^*(s'')$ need not be defined when the agent knows it can never reach states s'' given its horizon.

Boutilier et al. [11] showed how to determine an optimal policy by searching a decision tree; the *decision tree rollback procedure*[5]. The procedure is one of the *state-based search* approaches [11] to solve MDPs[6].

To determine which actions the agent should perform, one expands the state-transition diagram into a *decision tree*. If a robot is expected to perform N actions, a decision tree of depth N is generated. Figure 2.2 is the expansion of Figure 2.1 to horizon 2. In Figure 2.2, squares represent decision nodes, that is, at these nodes, the agent can 'choose an action' or 'make a decision'. The numbers in the squares indicate which state the agent will be in if it reaches that point (via the unique path) in the tree. Circles are chance nodes, that is, certain events occur, each with a probability, such that any one event at one chance node will definitely happen (probabilities of branches leaving a chance node, sum to 1).

From the decision tree, a *policy tree* is generated, that provides the robot with advice on the best action to take, for any state it may reach, for N actions in succession. If a policy tree for N actions is sought, we say a policy tree of horizon h is sought, where $h = N$. Figure 2.3 shows a

[5]Clemen and Reilly [20] explain in detail the method from the perspective of Decision Analysis.

[6]*Value* and *policy iteration* [88, 11] are two traditional, so called 'dynamic programming' approaches [11] to solve MDPs, but these are outside the scope of this dissertation.

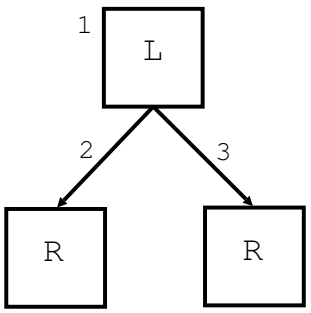

Figure 2.3: A policy tree recommending two actions (horizon of 2).

policy tree of horizon 2, derived from the decision tree of Figure 2.2.

The numbers '1', '2' and '3' in the figure identify the state the agent observes. Recall that in MDPs, full observability is assumed, which means that an agent modeled as an MDP always assumes that it identifies its current state correctly. Initially the agent is in state 1 and thus there is a '1' next to the root decision node. The initial action recommended is L. After performing L, the policy in Figure 2.3 tells the robot to execute R if it observes state 2 or to execute R if it observes state 3.

For the following example, abbreviate state 1 as s_1, state 2 as s_2 and state 3 as s_3. Let the immediate reward of each state be defined by a reward function as follows: $\mathcal{R}(s_i) = i$ —for simplicity. Each state can be reached with a certain probability (depending on the available actions and their outcomes). Using state rewards and probabilities, each path from the root to a leaf node determines a utility value (total expected reward) for that path. The path of N actions resulting in the highest utility value must be chosen. But the path cannot be chosen prior to execution time, because the outcome of any action is not known before an intended action is executed.

The decision tree rollback procedure is now presented in detail. To illustrate the procedure, the policy tree of Figure 2.3 is determined. The squares that are the leaf nodes in Figure 2.2 are decision nodes, but the agent will already have performed two actions by the time it reached a leaf node (leaf nodes represent states, not points of decision). So the last place an agent can actually make a choice is in one of the decision nodes in the second last tier of decision nodes (one of the four squares in this example). In this example, when at one of these last decision points, the agent must choose to go left or right; the agent has one more action to perform before its task is complete. No matter what rewards the agent has collected until this pre-last step, it must greedily choose the action that will (finally) add the maximum expected reward. Hence, if the robot first chose to move, say, left from the initial state and it nondeterministically ended up in state 2 (the outcome was that it moved left), then it will get a reward of value 2 plus what it *expects* to get after performing its next chosen action. But what *should* the robot choose to do when in its initial state; move left or right? In this example, the robot should move left because it can expect (probabilistically) to gain the most rewards by doing so:

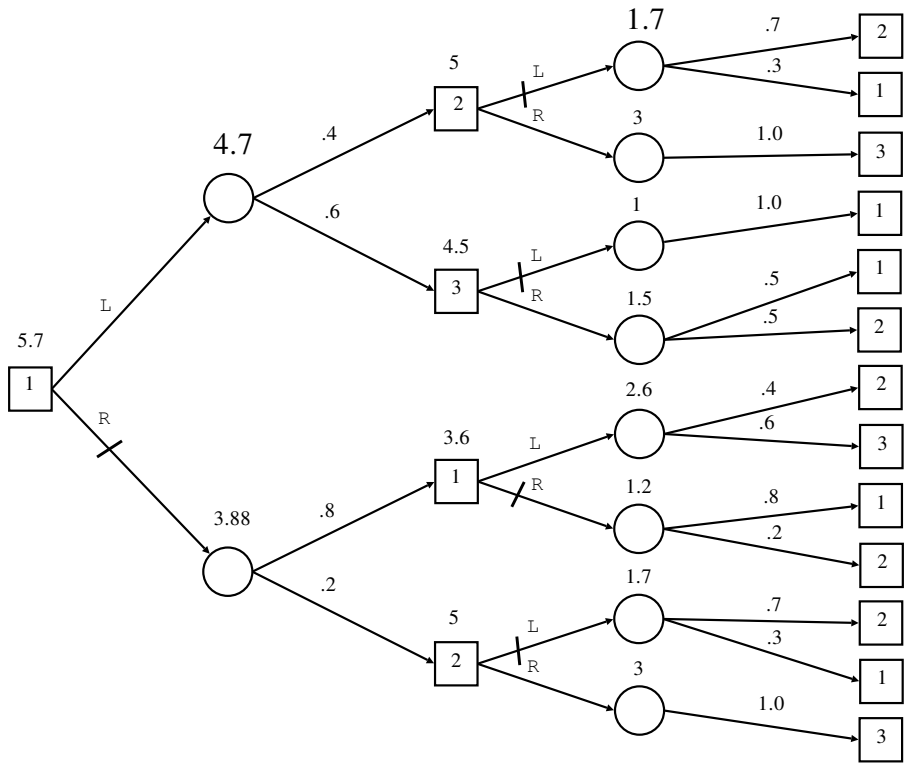

Figure 2.4: A decision tree showing probabilities of outcomes and expected rewards.

Figure 2.4 is the decision tree with information about the policy calculations shown. Crossed out arcs should not be chosen by the agent. Only one such arc per decision node will not be crossed out. Values above squares are total rewards expected to be gained after the 'maximizing' action is performed, plus the reward of the current state. Values above circles indicate only future expected rewards.

To conclude, using the example system behavior represented by Figure 2.1, the decision tree rollback procedure is as follows: Use the reward function to find the reward of each of the leaf nodes (squares in the last tier). Calculate the values above each circle in the last tier of the decision tree. For example, the 1.7—printed large in the figure—equals $0.7 \times 2 + 0.3 \times 1$. For each square in the preceding tier, pick the action that leads to the circle with the highest value. Add together this highest value and the reward of the state associated with the applicable square. Write this sum above the applicable square. Calculate the values above the two circles in the second-last tier. For example, the 4.7—printed large in the figure—is calculated as follows: $4.7 = 0.4(2 + 3) + 0.6(3 + 1.5)$. Pick *left*, because 4.7 is more than 3.88. Together with the reward of 1 for initially being in s_1, the agent executing the policy of Figure 2.3 can expect to have gained a total of 5.7 rewards (after 2 actions).

2.1.2 Partially Observable Markov Decision Processes

Again, mainly Russell and Norvig [88], Kaelbling et al. [48] and Boutilier et al. [11] were referenced for this subsection.

In partially observable Markov decision processes (POMDPs), actions have nondeterministic results as in (fully observable) MDPs, but observations are uncertain. In other words, the effect of some chosen action is somewhat unpredictable, yet may be predicted with a probability of occurrence. However, in POMDPs, the world is not directly observable: some data are observable and the agent infers how likely it is that the state of the world is in some specific state. The agent thus believes to some degree—for each possible state—that it is in that state, but it is never certain exactly which state it is in.

So, whereas in the MDP model, after any action, an agent will always know with complete certainty what state it ended up in, in the POMDP model, the agent only knows to a degree in what state it ends up. In fact, the agent maintains a probability distribution over the states to reflect the conviction it has that it is in a state (for each state).

The Model

Formally, a POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, b^0 \rangle$. $\mathcal{S}, \mathcal{A}, \mathcal{T}$ and $\mathcal{R}$ are defined as for MDPs. The other components of the POMDP model are:

- $\Omega = \{o_0, o_1, \ldots, o_m\}$ is a finite set of observations the agent can experience of its world; the observation at time t is denoted o^t; to be clear, an observation is not a sensing action, it is the information/data the agent has *after* or *provided by* each action with a sensory aspect;
- $\mathcal{O} : \mathcal{S} \times \mathcal{A} \to \Pi(\Omega)$ is the *observation function*, giving for each agent action and the resulting world state, a probability distribution over observations (we write $\mathcal{O}(s', a, o)$ to denote the probability of observing o in state s' resulting from performing action a in some other state; Russell and Norvig [88] write $\mathcal{O}(s, o)$, meaning the probability of observing o in state s); $\mathcal{O}$ represents the agent's trust in its observations, given the context of each observation;
- b^0 is the initial probability distribution over all world states in $\mathcal{S}$.

Kaelbling et al. [48] note that in the POMDP approach, there is no distinction made between actions to change the world (for instance, robot actuator activity) and information gathering actions (for instance, sensor activity of a robot). They mention that both kinds of action may have both kinds of effect.

Actuators (a) with no sensory feedback will have $\mathcal{O}(s', a, o) = 0$ for all s' and o, while actuators like servo motors that provide information about the motors' state, and sensors like sonars and IR detectors, could have, for example, $\mathcal{O}(s', a, o) = Pr_a(o|s')$, where Pr_a is the error profile of motor or sensor a.

Pineau [76] gives the *history* of the agent's activities as the sequence $(a_0, o_1, \ldots, o_{t-1}, a_{t-1}, o_t)$. Note that for every action there is an observation. Observations are not sensing actions; they are the data the agent has after an action. This implies that a 'pure locomotive' action will result in a *null* observation; only actions with *some* kind of information gathering component will result in *non-null* observations, and actions that are overtly sensory, will result in the greatest amount of observation. An example of a 'pure locomotive' action could be the activity of a robot's shoulder-joint actuator—the actuator does not produce feedback information and is thus a simple motor. A motor that supplies some data about its activity, and these data are captured by the robot's high level control system, would not be purely locomotive; such a motor has some sensing component, and actions produced by this motor would be associated with one or more non-null observations. An important function is the function that updates the agent's belief: Kaelbling et al. [48] call this function the *state estimation* function $SE(b, a, o)$. b is a set of pairs (s, p) where each state s is associated with a probability p, that is, b is a probability distribution over the set $\mathcal{S}$ of all states. b can be called a *belief state*. $SE(\cdot)$ is defined as

$$b^t(s') = \frac{\mathcal{O}(s', a, o) \sum_{s \in \mathcal{S}} \mathcal{T}(s, a, s') b^{t-1}(s)}{Pr(o \mid a, b)}, \tag{2.5}$$

where $b^t(s')$ is the probability of the agent being in state s' at time-step t.

Equation (2.5) is derived from the Baye s Rule. $Pr(o \mid a, b)$ in the denominator is a normalizer; it is constant with time. Note that $SE(\cdot)$ requires a belief distribution, an action and an observation as input; with every action there is an accompanying observation. $SE(\cdot)$ returns a new belief distribution for every action-observation pair. A procedural view of $SE(\cdot)$ could be as below.

Procedure *SE(o, a, b_{old})*

1 **forall** *states* $s \in \mathcal{S}$ **do**
2 $\quad b_{new}(s) = Pr(s \mid o, a, b_{old})$
3 **return** b_{new}

The state estimation function captures the Markov assumption: a new state of belief depends only on the immediately previous observation, action and state of belief.

Determining a Policy

Again, for any set of sequences of actions, the sequence of actions that results in the highest expected reward is preferred.

When the states an agent can be in are belief states, we need a reward function over belief states. We derive $\mathcal{R}b(b)$ from the reward function over world states, such that a reward is proportional to the probability of being in a world state. That is

$$\mathcal{R}b(b) = \sum_{s \in \mathcal{S}} \mathcal{R}(s) \times b(s).$$

Now the aim of using POMDP models is to determine recommendations of 'good' actions or decisions. Formally, a policy π, in POMDP theory, is a function from a set of belief states B (all those the agent can be in) to a set of actions:

$$\pi : B \rightarrow \mathcal{A}. \tag{2.6}$$

That is, actions are *conditioned* on beliefs. So given b^0, the first action a' is recommended by π. But what is the next belief state? This depends on the next observation. Therefore, for each observation associated with a', we need to consider a different belief state. Hence, the next action, a'', actually depends on the observations associated with and immediately after a'. In this sense, a policy can be represented as a *policy tree*, with nodes being actions and branches being observations. Function (2.6) is thus transformed to

$$\pi : \Omega \rightarrow \mathcal{A}. \tag{2.7}$$

In this sense, a policy is independent of agent beliefs.

The value function over *belief* states is defined as the *value* $Vb_{\pi,h}(b)$ of a belief state b with respect to a specific policy π. It is the expected sum of rewards for h steps and is defined by Equation 2.8:

$$Vb_{\pi,h}(b) = E\left[\sum_{t=0}^{h} \mathcal{R}b(b^t) \mid \pi, b^0 = b\right]. \tag{2.8}$$

Note that, on the right-hand side of Equation 2.8, the policy π is given. This implies that the value of function $\mathcal{R}b(\cdot)$ is affected by choices recommended by π at each step.

To find the expected probability of observations with respect to the next belief state b', a summation is performed over all observations—there is a new belief state for each observation considered (cf. Figure 2.6). That is, the POMDP *value function* is

$$\begin{aligned} Vb_{\pi,h}(b) &= \mathcal{R}b(b) + [\text{expected future rewards}] \\ &= \mathcal{R}b(b) + \sum_{o \in \Omega} [(\text{utility of belief state reached via } o) \times (\text{prob. of being in new belief state})] \\ &= \mathcal{R}b(b) + \sum_{o \in \Omega} Vb_{\pi,h-1}(b') Pr(o \mid a, b) \end{aligned}$$

where $b' = SE(o, a, b)$ and where the probability of being in the new belief state is $Pr(o \mid a, b)$, which is defined as

$$Pr(o \mid a, b) = \sum_{s' \in \mathcal{S}} O(s', a, o) \sum_{s \in \mathcal{S}} b(s) \mathcal{T}(s, a, s'). \tag{2.9}$$

$Pr(o \mid a, b)$ (cf. [19]) is the expected probability of reaching a new belief state s' taking into account all possible transitions from states in the old belief state, and the expected likelihood of observing o in s'.

Now the *optimal* policy π^* for a POMDP with planning horizon h (from the initial belief state) is definable:

$$\pi^* = \arg\max_{\pi}(Vb_{\pi,h}(b^0)). \tag{2.10}$$

This is the policy that will advise the agent to perform actions (given any defined observation) such that the agent gains maximum rewards (after h actions).

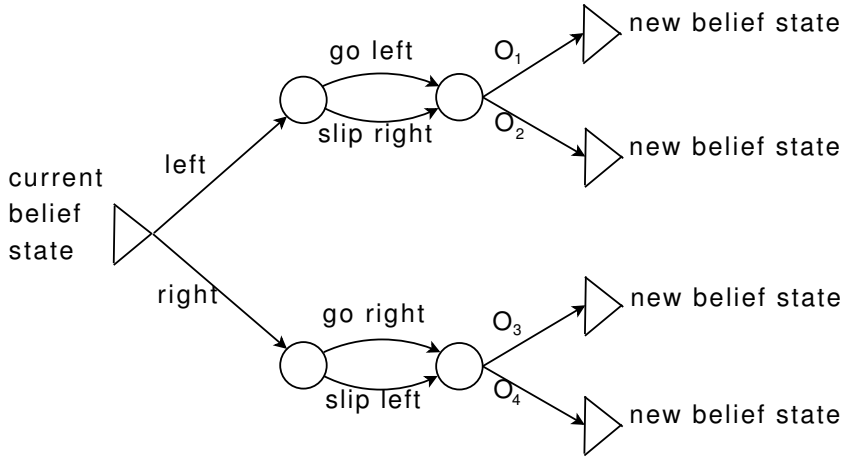

Figure 2.5: One tier of a belief-decision-tree.

To implement Equation (2.10), a *belief decision tree* is used[7]. An example sub-decision-tree (one tier) is shown in Figure 2.5. This example is based on an environment / agent model where the agent can only go left or right as depicted in Figure 2.1. Also, the agent may make two kinds of observations (O_1 and O_2) if it chose to go left, and another two kinds of observations (O_3 and O_4) if it chose to go right.

Belief states (triangles) in the belief-decision tree (simply 'decision tree' from now on) are decision nodes, and circles are again chance nodes.

The chance nodes reached first and the branches leaving them could have been left out of the diagram; they were included however, to illustrate the sequence of events that occur between consecutive belief states. The *go left, slip right, go right* and *slip left* branches are nature's realizations of the agent's decisions. The two pairs of action outcomes converge again because each single 'realization' does not determine a separate belief state. Expressed a different way,

[7] It is the expansion of a dynamic decision network (DDN) as in Russell and Norvig's book [88].

if the 'nature's choice' branches did split into separate subtrees, for the agent to make use of the decision tree for decision making purposes, it would need to determine (in the future) which action nature chose, so that the agent could know which subtree to follow. However, only *observations* bring in information. This is why there is a new belief state for each observation: the agent's beliefs turn out differently depending on what it observes. No matter what the observation, given the agent's intended action, the same whole set of nature's choices is considered in the calculation of the new belief state.

The discussion in the previous paragraph is reflected in Equation (2.5), where a new belief state is calculated for a *single* observation and for all realizations of the intended action. Notice that the summation is over all states, which in effect employs the state transition function for each state, that is, for all nature's choices.

A more conventional representation of Figure 2.5 is shown in Figure 2.6. It can be thought of as being modeled on a 'belief-MDP'. This is because it has the same structure as 'regular' MDPs, however, its states are not world states, but belief states.

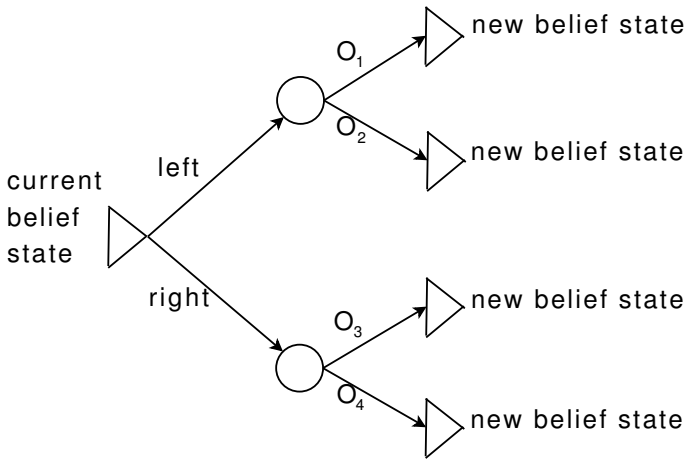

Figure 2.6: A conventional belief-decision-tree.

Employing the *rollback procedure* [11, 20], we roll back a decision tree to 'decide' the action. In any decision tree, for each action-observation pair, there is a tier of sub-decision-trees (such as in Figure 2.6). That is, when considering N actions in a row, a decision tree with N tiers would be required. There is a unique path from the initial decision node to each leaf node, and at each belief state encountered on a path, a reward is added, until (and including) the leaf belief state. At this point, the agent knows the total reward it would get for reaching that final state of belief. Each of the belief states is reachable with some probability.

At each decision node, a choice is committed to. Iteratively roll back—from last decision nodes to first decision node. The agent can in this way decide at the first decision node, what action to take. Each subtree rooted at the end of the branches representing the agent's potential action, has an associated expected reward. The action rooted at the subtree with the highest expected reward, should be chosen.

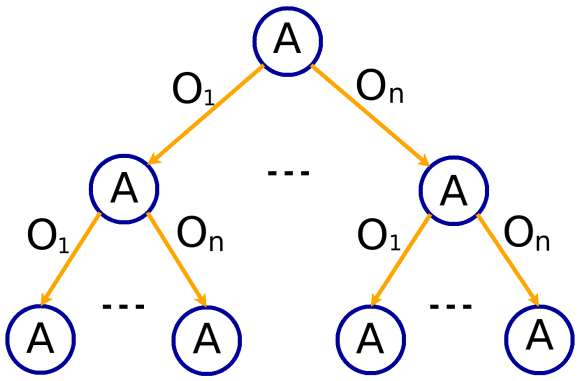

Figure 2.7: A generic representation of a policy tree for POMDPs.

An agent can choose only its actions (the best), not what it observes. Therefore, a policy recommends actions conditioned on observations. See Figure 2.7 for a generic representation of a POMDP policy tree. Note that a policy for POMDPs is conditioned on observations, not states as in MDPs. As the decision tree is rolled back, the best decision/action is placed into the policy, conditioned on the most recent possible observations. Using such a policy tree, the agent can always choose the appropriate action given its last observation. This is the essence of the theory on which the POMDP planner presented in Chapter 4 is based. An example policy calculation for an horizon of 1 can also be found in Chapter 4.

2.2 Decision Theoretic Golog

DTGolog = DT + Golog (DT stands for 'decision theoretic' here). The situation calculus is introduced first, because it is the basis of Golog. Then the Golog language and a decision theoretic dialect, DTGolog, are introduced.

2.2.1 The Situation Calculus

The situation calculus [66] is a first order logic (FOL) dialect for reasoning about dynamical systems based on agent actions. Actions and situations are *reified* to be objects in the language. The outcomes of a bout of reasoning in the situation calculus is meant to have an effect on the environment outside the agent. When an agent or robot performs an action, the truth value of certain predicates may change.

A special function symbol *do* is defined in the situation calculus. $do(a, s)$ is the name of the situation that results from doing action a in situation s. Note that $do(a_2, do(a_1, s))$ is also a situation term, where a_2 and a_1 are actions. In the situation calculus free variables are implicitly universally quantified.

For example, if a robot is holding a stone, the predicate $HoldingStone(r, s)$ (robot r is holding a

stone in situation s) is true, but when the robot does action $drop\_stone$, $HoldingStone(r, do(drop\_stone, s))$ should become false. Predicates and functions whose value can change due to actions are called *fluents*. Fluents have the *situation term* (s, $do(\cdot, s)$, etcetera) as the last argument.

To reason in the situation calculus, one needs to define an initial knowledge base (KB). The only situation term allowed in the initial KB is the special *initial situation* S_0. S_0 is the situation before any action has been done.

There are two more special formulae:

1. The *precondition axioms* are formulae of the form $Poss(a, s)$, which means action a is possible in situation s ($\neg Poss(a, s)$ means it is not possible). Precondition axioms need to be defined for each action. For example, if a robot r_{33} has only one arm and gripper, and the gripper is already holding a stone (in situation s'), that is, $HoldingStone(r_{33}, s')$ is true, then r_{33} cannot pick up something else; and then $\neg Poss(pick\_up(r_{33}, other\_stone), s')$ is true. The precondition axiom for the action $pick\_up(r, x)$ could be defined as follows:

$$Poss(pick\_up(r, x), s) \equiv \neg HoldingStone(r, s).$$

2. *Successor-state axioms* are formulae that define how fluents' values change due to actions. There needs to be a successor-state axiom for each fluent, and each such successor-state axiom mentions only the actions that have an effect on the particular fluent. Suppose there is one more action in our language: step_forward(r), meaning that robot r steps one step forward. Because stepping forward does not (usually) influence whether a robot is holding something, we could define the following successor-state axiom for the fluent $HoldingStone(r, s)$:

$$HoldingStone(r, do(a, s)) \equiv \exists x.a = pick\_up(r, x) \vee HoldingStone(r, s) \wedge a \neq pick\_up(r, x).$$

   Note that step_forward(r) is not mentioned in the above formula because it does not have an effect on the value of *HoldingStone*.

For reasoning in the situation calculus to be correct, the following is required:

- Σ, the four fundamental axioms for situations [84, Section 4.2]);
- $\mathcal{D}_{ss}$, all successor-state axioms;
- $\mathcal{D}_{ap}$, all action precondition axioms,
- $\mathcal{D}_{una}$, the set of unique names axioms for actions;

- $\mathcal{D}_{S_0}$, the set of formulas specifying the initial situation; and
- a formula that captures the *functional fluent consistency property* (refer to Reiter [84] for details).

Let $\mathcal{D} = \Sigma \cup \mathcal{D}_{ss} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$. Then $\mathcal{D}$ together with the functional fluent consistency property is the *basic action theory*.

Please refer to Reiter's book [84] for a detailed explication of his version of the situation calculus, including a description of the famous *frame problem* and how the successor-state axiom is a solution to this problem. Alternatively, refer to [14] for a one-chapter coverage of the situation calculus.

2.2.2 The Golog APL

Golog is an agent programming language (APL) developed by Levesque et al. [63]. It is based on the situation calculus. It has most of the constructs of regular procedural programming languages (iteration, conditionals, etcetera). What makes it different from software application programming languages is that it is used to specify *actions* and to control agent behavior via simple or complex constructs that are intended to be executed in the real world or a simulation of the real world. That is, Golog programs essentially constrain the search space of available actions, and provide a powerful means to specify and implement dynamical systems. Given a background action theory and an initial situation, an agent can directly execute an input program that specifies its expected behavior, that is, what the programmer expects the agent to do or how the programmer expects the agent to reach a goal.

Complex actions can be specified by combining atomic actions. The following are all complex actions (where a subscripted is an atomic action and φ is a formula):

- **while** φ **do** a_1 (iteration of actions);
- $\varphi\,?\,:a_1$ (test action);
- **if** φ **then** a_1 **else** a_2 (conditional actions);
- $a_1; a_2; \ldots; a_k$ (sequence of actions);
- $a_1 \mid a_2$ (nondeterministic choice of actions);
- $\wp x.(a_1)$ (nondeterministic finite choice of arguments—of x in a_1).

Let A be a complex action (also called a *program*) in the Golog language. Levesque et al. [63] defined the $Do(\cdot)$ macro procedure to interpret programs: $Do(A, s, s')$ holds if and only if the complex action A can terminate legally in s' when started in situation s.

Imagine a robot (*robot*5) that must collect stones one by one and deposit them on a predefined 'heap' location. But if the robot notices that its battery is running low, it must rather recharge its battery at the recharge station. Then *robot*5 must either look for a stone or get a new instruction from 'Control Center'. A program (P) to control *robot*5 might look like this:

[**if** $\neg BatteryLow(s)$

then $(HoldingStone(robot5, s)?$: [**while** $(\exists s')\neg AtHeap(s')$ **do** $gotoHeap; dropStone$])

else $(gotoRechargeStation;$ **while** $(\exists s'')BatteryLow(s'')$ **do** $charge)$];

[$lookForStone \mid getNewInstruction$].

Suppose S_{13} represents *robot*5's current situation and s' is a situation variable, then $Do(\cdot)$ can be used to find out what situation the robot will be in after executing program P from S_{13}: $\mathcal{D} \models Do(P, S_{13}, s')$ where $\mathcal{D}$ is the *basic action theory* for *robot*5. ($\models$ means *logically entails*.)

To illustrate how Golog can be employed for planning, suppose program P' is an uninstantiated variable. And suppose S_{13} and $\mathcal{D}$ are as before, and that S_g represents the situation in which the heap has ten stones on it: $HeapHas(10, Stones, S_g)$. Then if $\mathcal{D} \models Do(P', S_{13}, S_g)$, P' represents a plan for *robot*5 to get from S_{13} to S_g.

2.2.3 DTGolog

Decision-theoretic Golog (DTGolog) [13] is an extension of Golog to reason with probabilistic models of uncertain actions. The formal underlying model is that of fully observable Markov decision processes (MDPs)—a useful model in robotics, as most robots operate in environments where actions have uncertain outcomes.

The DTGolog interpreter however, does not simply 'perform' the program that it is given, but calculates an optimal policy based on an optimization theory: the decision tree rollback procedure for fully observable MDPs (cf. Sections 2.1.1 and 2.1.2).

In DTGolog, one captures the nondeterministic aspect of MDPs with the *stochastic* predicate and the *prob* predicate. $stochastic(a, s, n)$ determines the actual action performed in some situation s given action a was decided on by the agent; *stochastic* chooses from a finite set of deterministic actions. Thus DTGolog interprets the nondeterminism in a deterministic fashion. $prob(n, p, s)$ determines the probability p with which n is the outcome in s.

Further, to stay within the (fully observable) MDP framework, Soutchanski [100] specifies the $senseEffect(a)$ procedure that the agent uses to identify which action nature chose to perform for the robot instead of the robot's choice. $senseEffect(a)$ is defined by the system axiomatizer, and "these domain specific axioms define procedures that consist of one or a sequence of pre-

specified sensing actions that will be executed after a stochastic action," [100, p. 152]. This means that a DTGolog-agent is omniscient but not divining. The world is fully observable to an omniscient agent, but they cannot predict the outcome of their actions. Soutchanski [100] defines $choice'(a) \doteq \{n_1, \ldots, n_k\}$; $n_1 \ldots, n_k$ are the k actions[8] that nature could 'choose' for the agent's *intended* action a.

BestDo takes on the role of Golog's *Do*, but *BestDo* supplies an optimal conditional plan (policy). For stochastic actions,

$$\begin{aligned}
&BestDo(a; rest, s, h, \pi, v, pr) \doteq \\
&\exists \pi', v'.BestDoAux(choice'(a), a, rest, s, h, \pi', v', pr) \wedge \\
&\pi = a; SenseEffect(a), \pi' \wedge v = reward(s) + v'.
\end{aligned}$$

- $a; rest$ is the input program, with a the first action in the program and $rest$ the rest of the program.
- s is the situation term. When *BestDo* is initiated, s will be S_0, the initial situation.
- h is an input; the horizon of the policy that should be sought.
- π returns the policy.
- v is the accumulated value of rewards, used to choose the optimal conditional sequence of actions.
- pr returns the probability with which the input program will be executed as specified, given the policy and the effects of the environment.

BestDoAux takes care of nature's choices of actions deterministically (see below).

$$\begin{aligned}
&BestDoAux(\{n_1, \ldots, n_k\}, a, rest, s, h, \pi, v, pr) \doteq \\
&\neg Poss(n_1, s) \wedge BestDoAux(\{n_2, \ldots, n_k\}, a, rest, s, h, \pi, v, pr) \vee \\
&Poss(n_1, s) \wedge \exists \pi', v', pr'.BestDoAux(\{n_2, \ldots, n_k\}, a, rest, s, h, \pi', v', pr') \wedge \\
&\exists \pi_1, v_1, pr_1.BestDo(rest, do(n_1, s), h - 1, \pi_1, v_1, pr_1) \wedge \\
&SenseCond(n_1, \varphi_1) \wedge \pi = \textbf{if } \varphi_1 \textbf{ then } \pi_1 \textbf{ else } \pi' \textbf{ endif} \wedge \\
&v = v' + v_1 \cdot prob(n_1, a, s) \wedge pr = pr' + pr_1 \cdot prob(n_1, a, s).
\end{aligned}$$

For any action n, $SenseCond(n, \varphi)$ supplies a sentence φ that is placed in the policy being generated. φ holds if and only if the information supplied by the sensors due to $SenseEffect(a)$ can verify that action n was performed. In other words, φ is the 'conditional' part of the policy:

[8] Note that no situation term is involved in $choice'$: Soutchanski [100] also defines $choice(a, a_i, s)$ to hold in situation s if a_i is an outcome in nature of intended action a in s. In the sequel, only the simpler $choice'(a)$ is used.

after an action execution, an agent selects the next action conditioned on the truth of φ .

When either of two actions δ_1 and δ_2 can be performed, the policy associated with the action that produces the greater value (current sum of rewards) is preferred and that action is included in the determination of the final policy π. This formula captures the idea that is at the heart of the *expected value maximization* of decision theory:

$$
\begin{aligned}
&BestDo([\delta_1|\delta_2]; rest, s, h, \pi, v, pr) \doteq \\
&\quad \exists \pi_1, v_1, pr_1 . BestDo(\delta_1; rest, s, h, \pi_1, v_1, pr_1) \wedge \\
&\quad \exists \pi_2, v_2, pr_2 . BestDo(\delta_2; rest, s, h, \pi_2, v_2, pr_2) \wedge \\
&((v_1, \delta_1) \geq (v_2, \delta_2) \wedge \pi = \pi_1 \wedge v = v_1 \wedge pr = pr_1) \vee \\
&((v_1, \delta_1) < (v_2, \delta_2) \wedge \pi = \pi_2 \wedge v = v_2 \wedge pr = pr_2)).
\end{aligned}
$$

2.3 BDI Theory

The main components of the model for belief-desire-intention (BDI) agent architectures are: a set of beliefs, a set of desires, some intentions, plans that are more or less complex, and procedures for commitment to and reconsideration of intentions. Intentions are based on beliefs and desires, and plans are recipes of behavior to reach a goal or complete a task. Intentions are plans that have been committed to. Furthermore, future beliefs are influenced by past intentions, completing the circle.

It must be understood from the preceding paragraph that the components of the BDI model are not easily described one after another in a modular manner. Instead, the component concepts are co-dependent. This co-dependence means that the BDI model is better described in two passes, starting in general, somewhat informal terms, then in the second pass, defining the components formally—as far as is sufficient and necessary for this dissertation. In this section, due to the co-dependence of concepts and due to the two passes being made, some repetition is inevitable. Sections 2.3.1 and 2.3.2 constitute the first pass. Section 2.3.3 is the second pass, however, before heading straight into the formal body, the reader should first become familiar with key terms that will be referred to in later sections. The key terms are defined in Section 2.3.3. In Section 2.3.4, the reader will find some examples of existing implementations of BDI architectures.

2.3.1 Practical Reasoning and Folk Psychology

In folk psychology, we explain our actions with terms such as *know, think, believe, want, need, prefer, goal, desire, should, able to, impossible, intend, plan* and *act* or *action*. We infer our

mental state or mental attitude by observing our behavior and labeling the attitude with such a term. And we communicate our mental states using these terms.

We place much importance in the mind-set of a person when studying his/her actions. That is, we deem their *intentions* as important (that is, the person's focus of attention and commitments are deemed important). For example, when one hurts somebody on purpose—with the aim of inflicting pain—one acts with the *present intention* of hurting; 'acting with intent'.

We also talk about having intentions to do something in the future. This is a different kind of intention from present intention. *Future directed intention* concerns fixing on a goal to be achieved in the future.

In everyday language, we—folk psychologists—regularly use the concepts (mental states/ attitudes) of *belief*, *desire* and *intention* to talk about and explain our behavior. In the philosophy of *practical reasoning*[9], these three concepts are also used, amongst others. Each of belief, desire and intention (as mental states) is given much attention in practical reasoning. In fact, Bratman [15] put forward a theory of practical reasoning whose main elements are the *belief*, *desire*, *intention* and *plan*, to explain some aspects of human behavior. Bratman, in his theory, places much emphasis on especially intentions as related to plans.

Another philosopher, Dennett [25], expounds on the convention people have of viewing a complex system as having intentions. He proposes that we take such a view or 'stance' to facilitate our understanding of and reasoning about systems that are so complex that we cannot hold a *correct, clear* model of their internal processes. We might be able to take the *design stance* with relatively simple devices such as a kettle, a clutch-pencil or even a 1980s-type automobile. The design stance is to explain to ourselves how things behave by considering how they are designed. But when one thinks about a 2020s-type automobile, it may be so complex and seemingly with a degree of intelligence, that the design stance would fail one if one were to try explain the vehicle's behavior. The view that would most assist us in thinking about (explaining) the behavior of such an 'intelligent' vehicle, would be to subjectively assign intentions to it. That is, we could think of it as *having* intention. This is the *intentional stance*. Conceiving of complex systems in terms of intentions allows people to hide the confusing complexities of the system and allows people to reason about the systems in a quite efficient, compact manner. Software agents and autonomous robots can be some of the most complex systems in existence, and are thus prime candidates for us to take the intentional stance towards.

Mentioned here is the philosophical and psychological seeds of a theory of practical reasoning for agents. Bratman's theory [15] asserts the importance of intentions for planning of behavior and Dennett's intentional stance theory [25] adds veracity to designing and reasoning about agents with intentions. The next section introduces and informally discusses the BDI theory of

[9] *Practical reasoning* is reasoning about people's physical actions, whereas *theoretical reasoning* is reasoning about people's knowledge [114].

practical reasoning.

2.3.2 BDI Theory—Informally

Bratman [15] has developed a theory of practical reasoning that includes intention as a distinct mental state; distinct from the mental states of belief and desire. He argues for the necessity of intentions to influence and constrain plans and planning in intelligent but limited agents. Bratman accounts for present-directed and future-directed intentions, that is, acting with intention and intending to act in the future. In particular, he shows that future-directed intentions cannot be reduced to "clusters of beliefs and desires", and he says that a person's intentions involve a special commitment to action that ordinary desires do not. He says that desires and intentions are *pro-attitudes*—they inspire a volition to act, however, "intentions are, whereas ordinary desires are not, conduct-*controlling* pro-attitudes. Ordinary desires, in contrast, are merely *potential influencers* of action," [15, p. 16].

Rao and Georgeff [82] give another justification and explanation for the need for intentions. They say that beliefs are needed as the 'informative' component of a system state. Desires provide objectives for the system to accomplish; the 'motivational' state of the system. To control the balance between always replanning and never replanning (until the whole plan has been executed), the system must represent the "currently chosen course of action" called the system's intention: the 'deliberative' component of the system state.

Computational systems such as BDI agents "provide the essential components necessary to cope with the real world," [38, p. 2]—a world that is changing, where access to information is partial and where uncertainty prevails. Georgeff [38] explains why these components are essential: Belief is necessary for the usual reasons of the necessity for representation—the ability to keep information that is not directly perceivable and to use this information to make more effective decisions. A desire is thought of as a goal. Georgeff sets off goal-orientated computation against task-oriented computation. Task-oriented computation "is executed without any memory of what is being executed," [38, p. 4]. This means that such computation cannot easily recover from failure and cannot take advantage of opportunities. Decision theory is goal-oriented and when failure occurs; traditional decision theory will replan as soon as the failure occurs. The other extreme—as seen in task-oriented computation—is never to replan, but to continue attempting to execute the plan.

Practical reasoning can be divided into deciding *what* to do and determining *how* to do it [15]. Wooldridge [114] calls these two processes *deliberation*[10] and *means-ends reasoning* respectively. In the context of practical reasoning, *deliberation* means deciding on goals to pursue

[10] 'Deliberation' in robot architectures and 'deliberation' in BDI agent models mean different things. Deliberation in BDI models does not include planning, general problem solving nor meta-level control.

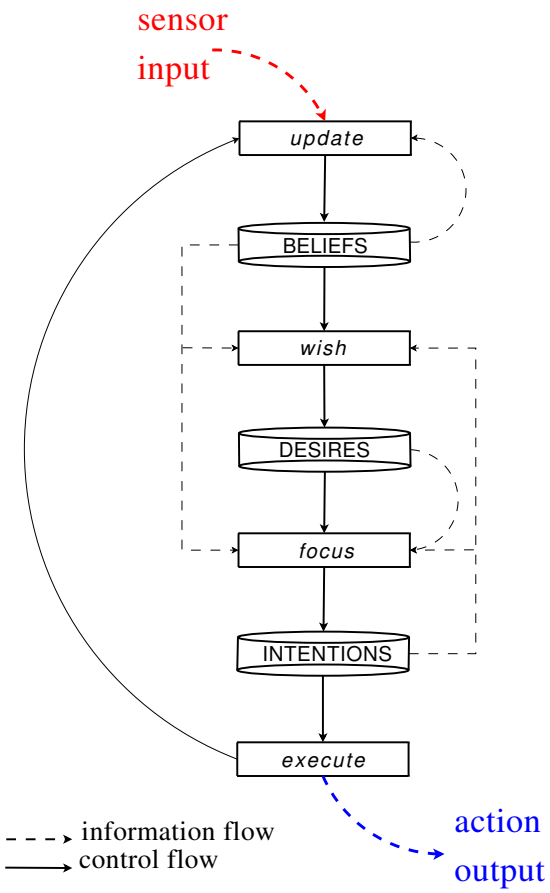

Figure 2.8: Schematic diagram of the generic belief-desire-intention model.

and *means-ends reasoning* means determining plans to achieve those goals (for now, consider *means-ends reasoning* and *planning* as synonyms). These two processes are computational, and as such, they require computer power, time and memory [113]. Any agent has a limit on the amount of computing power and memory, and because this work's focus is on agents living in the real world, they have real-time constraints. The agent thus has a limit on the time available to it before action must be taken. Agents with these limits are called *resource-bounded agents* [16].

A plan-based agent could reconsider its goals and plans after every update of its beliefs to keep its plans relevant. This may however lead to always thinking and never acting. On the other extreme, an agent could always act out its plans and never rethink its planned actions. This leads to the agent executing a plan that will be invalid with increasing likelihood as time passes. If intentions were to be fanatically followed, this would reduce to the case of never rethinking/reconsidering its intentions. On the other hand, an intention that is not committed to, that is, if it does not persist, it was not really an intention in the first place [113]. This is where commitment and reconsideration strategies come into play—they tell the agent when to consider new goals and plans for goals, that is, when to reconsider its intentions [15, 113]. But without intentions, there is nothing to reconsider. Intentions as commitments to some courses of action are thus essential for an agent to even attempt balancing its thinking and acting.

Figure 2.8 shows the model of an abstract BDI agent (adapted from Wooldridge [111]). *update* updates the agent's beliefs, *wish* creates a 'wish-list' of desires to pursue and *focus* decides on a subset of desires to seriously pursue as goals. More detailed models shall be seen in later sections.

To reiterate and clarify, *practical reasoning* is the *decision-making* or *thinking* of rational agents. The two main components of practical reasoning are reasoning about goals (goal selection) and reasoning about plans (goal pursuit). The real world changes, so replanning in cases of plans becoming invalid is essential. However, we would like to control *when* to replan, that is, we would like a mechanism to decide the level of commitment to plans. Intentions are plans that have been committed to. With the concept of an *intention* available to an agent, it can more easily reason about intentions, periodically dropping invalid intentions and instantiating new intentions as necessary in the current dynamic situation, with the situation's particular rate of change [38]. A system designed with beliefs, desires and intentions, together with plans "as a special kind of Belief" is what is known as a BDI agent [38].

Intentions can be formed, maintained and modified [81, 24]. Intentions are *maintained* by means of a *commitment strategy*. Intention maintenance concerns the commitment to *single* intentions that have already been adopted; whether an intention should be abandoned or maintained. Intention *modification* happens "in the light of changing circumstances," [81] through *re-deliberation*. An intention is scrutinized as to its applicability in light of new information. Maintenance and modification of intentions will be covered in Section 2.3.3.

Next is a more detailed discussion of plans in the light of BDI theory. Then, in Section 2.3.2, intention *formation* is introduced and further motivation is provided for why intentions are a central concept in the BDI model, and the roles that intentions play in the model are mentioned.

Partial Plans

Any intelligent agent will utilize plans. Plans have at least two uses, to reason whether a goal is achievable and as a recipe for achieving a goal. One way to view a plan for BDI agents is as a (pre-written) recipe. Such a plan to achieve an intention has a post-condition that matches/unifies with the intention, that is, with the goal/objective of the intention, and a pre-condition that is satisfied by the agent's current beliefs. An intention is often represented as a ground atom of first-order logic [113].

A partial plan is a coordinated set of abstract or vague activities. Such activities need to be 'filled in' with detailed plans—structured primitive actions. According to Bratman, we are not frictionless deliberators [15]: An agent with limited capabilities cannot, without inordinate cost, constantly redetermine "what would be the best thing to do in the present, given an updated assessment of the likelihood of" [15, p. 28] its own and other agents' future actions. If we create

concrete plans about the future, in many cases such plans become inapplicable. We rather settle on partial plans in advance and only reconsider them when we run into problems. So doing, we save our planning and deliberation resources for dealing with specific situations reactively and for attending to goals. Thus people use partial plans extensively.

There are two important demands on partial plans, (1) plans in general must be consistent, both internally and with the agent's beliefs, and (2) partial plans must be means-ends coherent, that is, relevant parts of the plan must be sufficiently filled in in time for the plan to be executed successfully. The abstract subplans can be filled in only when they are needed and thus when they can make use of current knowledge (knowledge that will be current in the future, and applicable to the subplan being filled in). Because intentions are similar to plans, intentions must also adhere to these two demands.

In BDI models, an agent must be capable of doing reasoning over *existent partial* plans, besides means-ends reasoning for plan *generation*. This entails knowing when and how to add details to partial plans. Pollack says

> [...] agents must be capable of committing to partial plans. If they were required always to form complete plans, they would over-commit, and filter out too many subsequent options as incompatible. But this then entails that [...] agents must have a way of deciding when to add detail to their existing plans—when to commit to particular expansions of their partial plans. [38, p. 7]

In the cognitive processes of a human, the distinction between what is an end (objective) and what is a means to the end (plan) is not clear. For instance, a means for achieving some end may be a sequence of lower-level or finer grained ends. And these lower-level ends could possibly be refined into sequences of ends that are at even lower levels. Now, one can see that an end *represents* a means, or viewed differently, there are no means, only hierarchies of ends. So when Bratman says that an intention is a commitment to a course of action, he is not committing to whether the intention is a means or an end: we "frequently reason from [...] a prior intention to *further* intentions [....] from more general to more specific intentions," [15, p. 17]. Ignoring whether an intention is a commitment to means or ends is troublesome in practice. Hence, in Section 2.3.3, this aspect of commitment will be formalized.

Desires and Intentions

Intentions are plans or goals that have been committed to.

Intention formation is *deliberation* proper, that is, it is intention *selection*. Intention maintenance and modification do not concern selection. The formation of intentions is the job of deliberation. Deliberation can be divided into two processes: option generation and filtering of goals [113]. It will be argued later that this definition is not adequate in general.

Desires need not be mutually consistent nor consistent with the agent's beliefs. Also, desiring a goal puts no demand on the agent to "settle on some means" to achieve the goal [15].

Bratman [15] distinguishes between a plan as an abstract structure, and a plan as a mental state. The former can be thought of as a procedure in memory that may be called upon when needed; the latter kind involves "an appropriate sort of commitment to actions," [15, p. 29] as in 'Don't worry, I have a plan.' Plans understood as mental states are "intentions writ large," according to Bratman. Intentions naturally apply to the coordination of partial plans, because partial plans have to include reasoning about the future, and (future directed) intentions concern future reasoning.

"Intentions are the building blocks of larger plans," [15, p. 32]. This does not mean though, that all intentions are formed through the process of planning—an intention may be formed by a separate psychological process, for example, coming across a picture in a book that reminds one of (awakens a) desire.

Prior intentions and plans provide reasons for actions in a different way to how beliefs and desires do [15]: intentions provide a framework for the choice of options via "demands for coherence and consistency", they do not provide input to the decisions for *weighing* alternatives—such weights are provided by beliefs and desires. A rational person will attempt to make her 'later'/'derived' intentions consistent with her 'prior' intentions, consistent with one another and consistent with her beliefs [15].

Intentions provide control in the BDI model in that they allow the agent to be reasonable or effective by balancing deliberation and action: An agent should commit to a course of action at some point, and then devote resources to achieving the course of action. Such a commitment to a course of action is what we call an intention.

According to Pollack "Bratman's Claim" is that "rational agents will tend to focus their practical reasoning on the intentions they have already adopted, and will tend to bypass full consideration of options that conflict with those intentions," [38, p. 6]. As support for Bratman's Claim, it explains one way how agents can act reasonably with bounded resources: a reasonable resource-bounded agent will "avoid getting lost in the morass of options for actions available to it" [38, p. 6] by some strategy for committing to plans.

Pollack [38, p. 6] distinguishes between models that employ the folk-psychology concepts of belief, desire, and intention: those that do and those that do not incorporate Bratman's Claim. In this dissertation, when talking about BDI models, we shall mean those that incorporate the claim.

In combination, future-directed intentions and partial plans help support coordination and extend the influence of practical reasoning over time [15]. Further, Bratman argues that the idea of future-directed intentions implies an element of commitment, but that they are not *irrevoca-*

ble. If an intention to act were irrevocable, it could just as well be decided at the moment of the act, making intention (about the future) a waste of time. But then comes the question, if all intentions are revocable, are they worth anything? And more, if they are worth something, when should a rational agent reconsider its commitment to plans and intentions?

Wooldridge [113, 114] summarizes the roles of intentions as follows.

- Intentions drive means-ends reasoning. They are pro-attitudes: if I have an intention, I will formulate plans to achieve it. "They tend to lead to action," [114, p. 67]. And in this sense, they are stronger than the pro-attitude of desire, because desires are "merely potential influencers," [114].
- Intentions persist. Intending implies committing for an *extended period*. However, an intention should be dropped by a rational agent if it is believed already achieved, believed unachievable or believed that its post-condition is no longer required. An agent will keep on trying to achieve an intention for as long as it is reasonable, even after failure to achieve it.
- Intentions constrain future deliberation. An agent should not entertain options (desires) that are inconsistent with current intentions. In other words, a reasonable agent should not adopt a new intention if it conflicts with existing intentions. And existing intentions have preference because they are persistent.
- Intentions influence beliefs upon which future practical reasoning is based. Intentions must be consistent with beliefs. Therefore, if the situation changes such that some intention could become inconsistent with some belief, the agent may decide not to update its beliefs in such a way, instead of abandoning the conflicting intention. If an agent believes that a situation may occur in which an intention will be satisfied and at the same time the agent believes there is another situation in which the intention cannot be satisfied, it is called belief-intention incompleteness. If an agent believes that no situation may occur in which the intention will be satisfied, yet adopts or maintains the intention, it is called belief-intention inconsistency.

2.3.3 A Formal Model of BDI Agency

Sorting-out Terms

To clarify concepts, the following terms are now formally defined *motivation, desire, goal, o-intention, p-intention, recipe, policy* and *plan*. Terms *planning, deliberation* and *achievement* are also clarified.

A *motivation* is a (pre-programmed) specification of one element of an agent's innate drive.

Together, all an agent's motivations determine all the states (ends) the agent would ideally like to be in over its lifetime. Motivations may not be directly accessible by the agent, and are typically regarded as part of the agent's background knowledge base. In this work, an agent's motivations are assumed not to change—they remain fixed during its lifetime.

As discussed before, there are ends/objectives/goals and means/procedures/recipes for achieving them. We shall call the former *objectives* and the latter *plans*. Hence, an *objective* is a *reference* to a desired state and a *plan* is a structure of actions and rules for how/when to execute the actions.

Desires are a subset of objectives—as specified by motivations—that an agent would ideally like to pursue/achieve, according to its *current* beliefs. Desires need not be mutually consistent nor consistent with the agent's beliefs. Also, desiring a state puts no demand on the agent to "settle on some means" to achieve the state [15].

"Although, in the general case, desires can be inconsistent with one another, we require that *goals* be consistent. In other words, goals are chosen desires of the agent that are consistent. Moreover, the agent should believe that the goal is achievable. This prevents the agent from adopting goals that she believes are unachievable and is one of the distinguishing properties of goals as opposed to desires. Cohen and Levesque [Cohen and Levesque, 1987] call this the property of *realism*," [81, pp. 2–3].

An agent will deliberate to choose a subset of desires, which are then its set of goals. While it deliberates, it knows that it will select a subset of these goals to seriously pursue. An *o-intention* (*objective-intention*) is a goal that has been selected—committed to, according to some policy, strategy or value judgment. Desires, goals and o-intentions are all *objectives*. Their *plans* must be determined, calculated or selected separately.

Recipes are pre-compiled or pre-assembled *plans*, usually stored in a 'plan library'. We follow De Silva and Padgham in that the term *recipe* shall be used to refer to a pre-compiled plan; "pieces of code the programmer writes," [24]. A *policy* is a plan produced by a planning program, that is, a policy is a generated plan. The term *plan* shall be left to refer to plans in general, including recipes and policies or any (conditional) sequence of actions. Plans are behavioral recipes or policies that an agent has or is capable of generating. Every plan achieves one or more of the agent's desires. More than one plan may achieve the same desire.

There can be commitment to plans (means) and to objectives (ends) [113]. This is evident in human cognition. People commit to (achieving) a desire *and* they commit to some way (plan) of achieving what they have decided they really want. A plan to achieve a committed-to desire is a *p-intention* (*plan-intention*). The presence of a p-intention implies the presence of an o-intention, but not necessarily vice versa. For example, I may have the o-intention of obtaining my Masters degree and I may have two options for achieving this goal: full-time study for

two years, or part-time study for five years. Once I have committed to either the full-time or part-time option (p-intentions), I have—by logical reason—committed to obtaining my Masters (o-intention). However, committing to obtaining my Masters degree (o-intention) does not force me to immediately commit to an associated plan of action (p-intentions). An *intention* is an o- or p-intention.

Planning is the process of determining (selecting from a library or generating with a planner) a single p-intention for a single o-intention. As soon as a plan is selected to achieve an o-intention, the plan becomes a p-intention by definition. In the BDI model each objective may have several plans for achieving it. So while an agent can keep an o-intention in mind, it can choose the most effective p-intention to gain its vision.

Deliberation is the process of determining a set of o-intentions (*what* to achieve) according to some rules or value judgments. We could say, if a process seeks a recipe or policy for achieving an objective (that is, if planning is involved), it is not part of deliberation, however, see the discussion in Section 2.3.3. Wooldridge [113] divides deliberation into two processes: (i) generation of a set of desires, and (ii) filtering (selecting) o-intentions from the set of desires.

Achievement of an o-intention means that the specification of the o-intention is satisfied in the agent's current state, whether its associated p-intention has been completed or not. *Achievement* of a p-intention means that the sequence of actions or the conditional plan that makes up the p-intention has been executed to completion. When a p-intention has been achieved, its associated o-intention has been achieved.

Summarizing, we defined *motivations* (innate driving force), *objectives* (references to preferred states), *plans* (either pre-compiled or generated), *desires* (all current objectives), *goals* (a subset of consistent and possible desires), *o-intentions* (desires committed to), *recipes* (pre-compiled, hand-written plans), *policies* (generated plans) and *p-intentions* (plans committed to). The terms *planning, deliberation* and *achievement* were also defined.

Discussion

BDI architectures have traditionally employed partial plans. A partial plan is a plan that is an aggregation of references to more detailed recipes of actions. These 'subplans' may also contain references to even more detailed subplans, and so on. An o-intention may thus be part of another o-intention. We insist that an o-intention is always a *reference* to a desired state of affairs, nothing more.

Let '$A \Longrightarrow B$' be a procedure, with A the input and B the output. Suppose some architecture follows this process: motivations $\Longrightarrow$ desires $\Longrightarrow$ goals $\Longrightarrow$ o-intentions $\Longrightarrow$ p-intentions.

A different approach may be to select the subset of *options* from the desires, instead of gen-

erating the goals from the desires. Options may not satisfy the requirement of being a goal set: perhaps options are selected according to a value judgment, and then consistency is sought when selecting intentions from the options. The process will then be: motivations $\Longrightarrow$ desires $\Longrightarrow$ options ($\Longrightarrow$ o-intentions) $\Longrightarrow$ p-intentions. "($\Longrightarrow$ o-intentions)" indicates that the architecture does not explicitly select o-intentions during deliberation, but rather more obviously selects p-intentions (as in, for example, PRS [47]).

The *option* or *desire* generation procedure plays several roles. Wooldridge [111] writes

> First, it must be responsible for the agent's means-ends reasoning—the process of deliberating how to achieve intentions. Thus, once an agent has formed an intention to x, it must subsequently consider options to *achieve* x. These options will be more concrete—less abstract—than x. As some of these options then become intentions themselves, they will also feed back into option generation, resulting in yet more concrete options being generated. We can thus think of a BDI agent's option generation process as one of recursively elaborating a hierarchical plan structure, considering and committing to progressively more specific intentions, until finally it reaches the intentions that correspond to immediately executable actions. [p. 59]

Options are not plans to achieve intentions, but are a set of objectives that the agent can choose from to achieve. Selected options become o-intentions. A plan function separate from option generation then does planning to determine how to achieve o-intentions.

The reader's attention is drawn to two caveats: (1) the fact that a BDI architecture does not always follow the process ...desires $\Longrightarrow$ goals $\Longrightarrow$ o-intentions... and that (2) considering the process ...options $\Longrightarrow$ p-intentions... one would suspect that means-ends reasoning (planning) must have taken place during the procedure ($\Longrightarrow$) to determine the set of plans committed to (the p-intentions).

Hendriks et al. [43] make the point: APLs (for *implementing* agents) like Agent-0, AgentSpeak, 3APL and ConGolog do not have goals as declarative concepts. Their goals and intentions are plans ("structures built from actions"). Georgeff and Lansky say about their Procedural Reasoning System, "Unlike most AI planning systems, PRS goals represent desired behaviors of the system, rather than static world states that are to be [eventually] achieved," [40, p. 679]. This is in contrast to the logical systems for specifying and reasoning *about* agents (not *implementing* them). BDI architectures that have intentions as *declared goals* are: an extension to 3APL [22] and the CAN [110] and EAGLE [56] programming languages. How, if at all, is serendipity taken advantage of without declared (declarative) goals? One must know what state one is aiming for to know if one fortuitously achieved it before one's plan is completed. The question is outside the scope of this dissertation.

In conclusion, a formal definition of *deliberation* within the BDI model is highly dependent on

the architectural realization of the model and cannot be properly defined for the model in general. The definitions of terms given in Section 2.3.3 must thus be taken as 'working' definitions.

Initially the agent's o-intention set is empty and it may happen that the set again becomes empty after some time. Whenever the agent has no intentions, it needs to adopt one or more new intentions.

If the agent is instantiated with an empty o-intention set, the agent designer (or the agent architecture) determines how many o-intentions to place in the set. Theoretically, one o-intention is enough for an agent to become active. The agent can always keep one o-intention by deliberating as soon as the old one is dropped for some reason. At the other extreme, the agent could adopt all its goals as o-intentions. A set of two or more o-intentions could be ordered in a stack according to a metric of value or urgency, or if the implementation system allows, the agent could pursue some or all of its intentions concurrently. The number of o-intentions selected during deliberation may also vary according to the agents beliefs and available resources.

But why *should* an agent re-deliberate to refill an empty intention set? After all, when the intention set has *become* empty, does it not mean that the agent has satisfied and achieved all its intentions? The reason is that normal reasonable agents, including reasonable people, have *desires* that are never satisfied. Some kinds of desires are satisfied, but a human being will always have a few 'fundamental' desires from which all other desires are derived, and which give the person his/her fundamental/initial/innate/original motivation and drive. Once a person decides what desires to pursue (via goals), the desires become o-intentions and the o- and p-intentions together can be achieved. However, a fundamental desire can never be achieved, by definition. For example, we never have enough wealth, knowledge, love or peace.

We adopt plans consisting of sub-plans (sub-intentions; lower level o- or p-intentions, depending on your view) to temporarily satisfy our desires. Once we have achieved all the sub-plans of a particular plan, we will only shift our focus to satisfy a different desire for a period. When we have satisfied this new desire by achieving some sub-intentions, we re-deliberate and choose the next important desire to satisfy. Hence only top-level o-intentions are ever derived from / filtered out of the desire set; lower-level o-intentions are never derived from desires—they become active only because they are part of a (sub)plan that has been committed to. Therefore, deliberation involves finding only top-level o-intentions.

For completeness, we define two *depths of deliberation*, goal-deep and intention-deep. *Goal-deep* deliberation generates goals from innate desires, then selects some o-intention(s) from the new goal set. *Intention-deep* deliberation does not generate new goals; it selects some o-intention(s) from the old goal set.

The Formal Model

Since Bratman's book [15], scientists and philosophers (including Bratman) have 'distilled' the original BDI theories. This process has involved experimentation and has been guided by the framework of computer science to the point where a BDI agent is defined in computational and formal terms, and hence does not match the original theory exactly. Indeed, the original theory is based on a philosophy of *human* practical reasoning, whereas computational BDI agents are artificial and could not hope to be a simulation of human behavior at this time. Moreover, the BDI model of agency and, in fact, all agent theories are still relatively new; no standard definition or categorization of agents has yet been agreed upon. In this and the next subsections, the BDI theory presented earlier is formalized.

Formally, a BDI agent has at least these seven components [111] (cf. Figure 2.9):

- B, a knowledge base of beliefs;
- An option generation function, generating the objectives the agent would ideally like to pursue (its desires). Call the function *wish*;
- D, a set of desires returned by the *wish* function;
- A function that filters out incompatible, impossible and less valuable desires, and that focuses on a subset of the desire set. Call the function *focus*;
- I, a structure of intentions; the most desirable options/desires returned by the *focus* function;
- A belief change function: given the agent's current beliefs and the latest percept sensed, the belief change function—call it *update*—returns the updated beliefs of the agent;
- A function that selects some action(s) from the currently active plan and executes it. Call the function *execute*.

$wish : B \times I \rightarrow D$ generates a set of desires, given the agent's beliefs, current intentions and possibly its innate motives. It is usually impractical for an agent to pursue the achievement of all its desires. It must thus filter out the most valuable desires and desires that are believed possible to achieve. This is the function of $focus : B \times D \times I \rightarrow I$, taking beliefs, desires and current intentions as parameters. Together, the processes performed by *wish* and *focus* may be called deliberation, formally encapsulated by the *deliberate* procedure.

Algorithm 2 (adapted from Wooldridge [113, Fig. 2.3]) is less abstract than Figure 2.8 (Section 2.3.2). The functions *getPercept* and *plan* appear in Algorithm 2, but are hidden in Figure 2.8. *getPercept*() senses the environment and returns a percept (processed sensor data) which is an input to *update*(·); given the agent's intention structure and its beliefs, *plan*(·) selects an intention from the intention structure and returns a plan to achieve it.

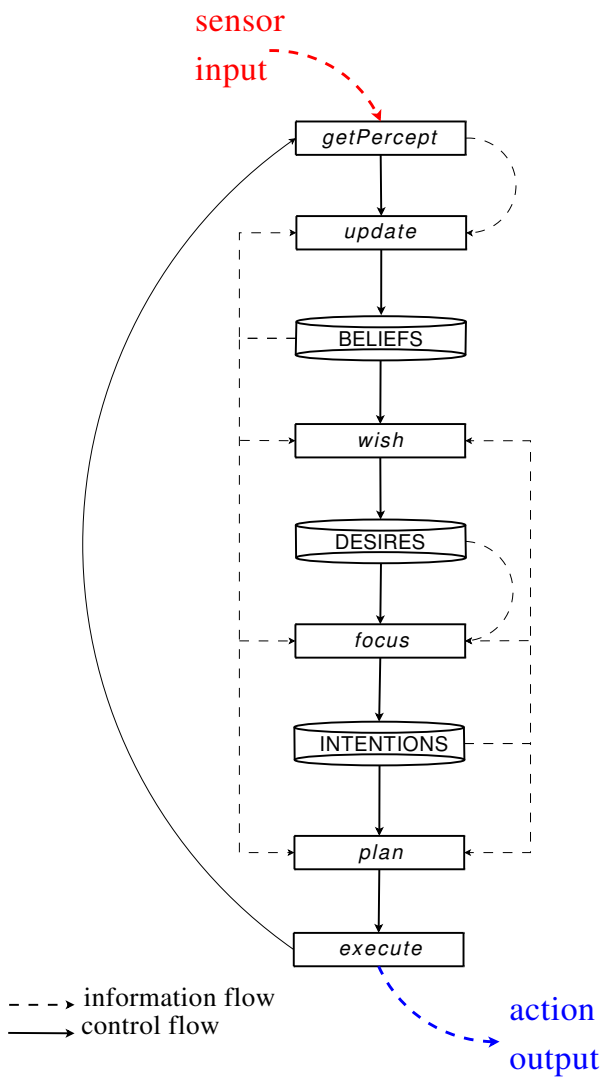

Figure 2.9: More detailed schematic diagram of the generic belief-desire-intention model.

Note that in Algorithm 2, the generation of desires is constrained by I, the selection of intentions is constrained by D and I, and finding a plan is constrained by I. This is in accordance with BDI theory: future reasoning (deliberation and planning) is constrained by past commitments (intentions).

Figure 2.9 reflects Algorithm 2 graphically.

Assume that an agent's planning capability is represented by a *plan* function. According to Wooldridge [113, p. 30]

> [...] there is nothing in the definition of the *plan*(...) function which requires an agent to engage in *plan generation*—constructing a plan from scratch [...] In most BDI systems, the *plan*(...) function is implemented by giving the agent a *plan library*. [p. 30]

Intentions can be formed, modified and maintained [81, 24]. Intention formation was covered in detail in Sections 2.3.3. In the following two sections, the latter two processes are discussed in detail.

Algorithm 2: Basic BDI agent control loop

Input: B_0: initial beliefs
Input: I_0: initial intentions

```
1  B ← B_0;
2  I ← I_0;
3  π ← null ;
4  while alive do
5      p ← getPercept();
6      B ← update(B, p);
7      D ← wish(B, I);
8      I ← focus(B, D, I);
9      π ← plan(B, I);
10     execute(π);
```

Intention Maintenance

A commitment strategy says when an individual o-intention or an individual p-intention should be maintained or abandoned; whether commitment should be continued or discontinued. A commitment strategy does not, directly involve a decision whether to deliberate, that is, whether to find a new set of o-intentions. It involved the decision whether to *keep or drop* an intention or part thereof.

> As the agent has no direct control over its beliefs and desires, there is no way that it can adopt or effectively realize a commitment strategy over these attitudes [beliefs and desires]. However, an agent can choose what to do with its intentions. [82, pp. 315–316]

Rao and Georgeff define three commitment strategies [81, 82, 103]:

- a *blindly committed* agent keeps an intention until believed achieved. The agent is over committed. It keeps on trying to achieve its intentions, whether they are believed possible or not. This is irrational. Also, changes to a blindly committed agent's desires (from which its intentions are derived) have no influence on its intentions;
- a *single-minded* agent keeps an intention until believed achieved or until believed impossible, acceding to any beliefs that would indicate the impossibility of achieving the commitments, that is, allowing changes in beliefs to cause it to drop some commitments, but remaining unaffected by changes in desires; and
- an *open-minded* agent keeps an intention until believed achieved or impossible or until it is no longer a desire. The agent is under committed because its intentions might be dropped due to changes in its beliefs *and* its desires. Hence, such an agent is influenced by its desires, and does not strongly commit to an intention once it has decided to commit to it. The agent is 'weak'.

The choice of commitment strategy is up to the agent designer and is informed by the application environment.

Algorithm 3: Control loop for a single-minded BDI agent with reactivity

Input: B_0: initial beliefs
Input: I_0: initial intentions
1 $B \leftarrow B_0$;
2 $I \leftarrow I_0$;
3 $\pi \leftarrow null$;
4 $p \leftarrow getPercept()$;
5 $B \leftarrow update(B, p)$;
6 $D \leftarrow wish(B, I)$;
7 $I \leftarrow focus(B, D, I)$;
8 **while** *alive* **do**
9 **if** *not empty*(π) **then**
10 **if** *not sound*(π, I, B) **then** $\pi \leftarrow plan(B, I)$;
11 $\alpha \leftarrow head(\pi)$;
12 $execute(\alpha)$;
13 $\pi \leftarrow tail(\pi)$;
14 $p \leftarrow getPercept()$;
15 $B \leftarrow update(B, p)$;
16 $I \leftarrow drop{-}succeeded(I, B)$;
17 $I \leftarrow drop{-}impossible(I, B)$;

Algorithm 3 is an agent controller for a single-minded agent (adapted from Wooldridge [113, Fig. 2.5] and Rao and Georgeff [82, p. 318]); we add one test that will enforce a single-minded commitment strategy. That is, the agent tests at every iteration through the main loop whether the currently pursued intention is still possibly achievable, using $impossible(\cdot)$. In the algorithm, serendipity is also taken advantage of by periodically testing—using $succeeded(\cdot)$—whether the intention has been achieved, without the plan being fully executed. We call this agent 'reactive' because the agent executes one action per loop iteration; this allows for deliberation between executions.

Lastly, the soundness of the plan to achieve the current intention is checked at every iteration of the loop. Informally, soundness of a plan is different from the achievability of an intention in that achievability concerns action *executability* and the existence of the goal state, whereas a sound plan must in *principle* recommend the correct actions, that is, a 'legal' sequence of actions, to achieve the goal state. Refer to Lifshitz [64] and Pollock [78] for articles on the subject of soundness of plans. Algorithm 3 includes $sound(\cdot)$ at line 10.

A plan will be abandoned when it is not sound. Intentions impossible to achieve or that have been achieved, will be dropped. Note that after the intention set is determined from the initial desire set D (before the loop), D no longer has an influence on the maintenance of intentions; D does not occur inside the loop.

Intention Modification

Whereas a commitment strategy says when an individual intention should be kept or dropped, a reconsideration strategy says when to deliberate. Reconsideration is equivalent to re-deliberation!

A *reconsideration strategy* is not a *commitment strategy*: Reconsideration does not merely involve the entertainment of a possible change in an intention; it seriously reopens "[...] the question of whether to *A*, so that this is now a matter that needs to be settled anew," [15, p. 62]. That is, when an agent decides to reconsider, it activates its deliberation process. The agent considers its reasons for forming an intention, it does not merely consider some intention(s) on the surface, as is the case when an agent employs its commitment strategy. Reconsideration is also described by, for example, Wooldridge [113] and "was examined by David Kinny and Michael Georgeff, in a number of experiments," [111, p. 57].

Intentions resist change, but there are conditions under which a rational agent *must* consider whether its intentions are still worth committing to. That is, an agent should reconsider its intentions when it is reasonable to do so.

Possible interactions between deliberation and meta-level control (whether to deliberate again) are summarized in Table 2.1 (copied from Wooldridge [113, p. 39]). The analysis elucidates when reconsideration is optimal. In situation 1, the agent does not deliberate, but if it did, it

Situation number	Chose to deliberate?	Changed intentions?	Would've changed intentions?	*reconsider*(·) optimal?
1	No	-	No	Yes
2	No	-	Yes	No
3	Yes	No	-	No
4	Yes	Yes	-	Yes

Table 2.1: Interactions between meta-level control and deliberation.

would not have changed its intentions anyway. This situation is desirable. In situation 2, the agent does not deliberate, but if it did, it *would* have changed its intentions. Here the agent gets bad advice from *reconsider*(·). The agent chooses to deliberate in situations 3 and 4. When it does not change its intentions in situation 3, the agent is wasting time deliberating. The *reconsider* function is not behaving optimally. The agent does change intentions in situation 4, which means it was a good idea to deliberate, and *reconsider*(·) has done well.

"Notice that there is an impotent assumption implicit in this discussion: that the cost of executing the *reconsider* function is *much* less than the cost of the deliberation process itself," [113, p. 39]. If the cost of *reconsider*(·) were more than that of deliberation, the long term cost of employing *reconsider*(·) would be more than if it were not employed, given that *reconsider*(·) is run once every iteration of the control loop when employed, and *wish*(·) and *focus*(·) (deliberation) are run once every iteration of the control loop when meta-control is not employed.

Please refer to Wooldridge and Parsons [112] for an in-depth formal analysis of intention reconsideration in the spirit of the above discussion.

Algorithm 4: Control loop for a cautious agent

```
   Input: B_0: initial beliefs
   Input: I_0: initial intentions
1  B ← B_0 ;
2  I ← I_0 ;
3  π ← null ;
4  while alive do
5      p ← getPercept() ;
6      B ← update(B, p) ;
7      D ← wish(B, I) ;
8      I ← focus(B, D, I) ;
9      if not empty(π) then
10         if not sound(π, I, B) then π ← plan(B, I) ;
11         α ← head(π) ;
12         execute(α) ;
13         π ← tail(π) ;
14     I ← drop−succeeded(I, B) ;
15     I ← drop−impossible(I, B) ;
```

Algorithm 4 (adapted from Wooldridge [113, Fig. 2.6] and Rao and Georgeff [82, p. 318]) is for a cautious agent whose reconsideration strategy is to always re-deliberate; it reconsiders its intentions before every action it performs.

To control when the agent would consider *whether* to re-deliberate, the *reconsider* function is placed just before deliberation would take place (that is, before option generation and filtering; lines 7 and 8 in Algorithm 4), resulting in the agent defined by Algorithm 5 (adapted from Schut and Wooldridge [97]) and still has the single minded commitment strategy.

Realize that the presence of $reconsider(\cdot)$ at line 7 in Algorithm 5 does not mean that the agent reconsiders every time line 7 is reached; $reconsider(.)$ is a Boolean function that tells the agent *whether* to reconsider its intentions.

There are various mechanisms that an agent might use to decide when to reconsider its intentions. We shall call such a mechanism a *reconsideration strategy*. Four classes of reconsideration strategies are proposed in this dissertation: *periodical*, *conditional*, *knowledge-based* and *value-based*:

- The periodical strategy is to reconsider at fixed intervals. A cautious agent is defined by Algorithm 4 and it uses a periodical reconsideration strategy. A trivial periodical strategy is to *never* reconsider; if the *reconsider* function in Algorithm 5 were to always evaluate to *false*, it would define a bold agent. Pollack and Ringuette [77] first defined bold and cautious agents. Kinny and Georgeff [52, 53] did a series of experiments showing the

Algorithm 5: Control loop for an agent with reconsideration

Input: B_0: initial beliefs
Input: I_0: initial intentions

```
1  B ← B_0 ;
2  I ← I_0 ;
3  π ← null ;
4  while alive do
5      p ← getPercept() ;
6      B ← update(B, p) ;
7      if reconsider(B, I) then
8          D ← wish(B, I) ;
9          I ← focus(B, D, I) ;
10         if not sound(π, I, B) then π ← plan(B, I) ;
11     if not empty(π) then
12         α ← head(π) ;
13         execute(α) ;
14         π ← tail(π) ;
15     I ← drop−succeeded(I, B) ;
16     I ← drop−impossible(I, B) ;
```

effects of following bold and cautious strategies in different environments. Schut and Wooldridge define the *degree of boldness* as "the maximum number of plan steps the agent executes before reconsidering its intentions," [95, p. 210].

- The conditional strategy is to reconsider when some o-intention takes 'too long' to be achieved or when the o-intention set becomes empty.
- The knowledge-based strategy is to reconsider when some logical rule says to do so. For example, there may be a rule that says to reconsider whenever a percentage of o-intentions has become unachievable (impossible), or there may be a rule that says to reconsider whenever some p-intention has become unsound and there is no other (sound) p-intention that achieves its o-intention.
- The value-based strategy is to reconsider whenever a reconsideration policy recommends it. A reconsideration policy is either determined during run-time or before, and the policy is found via a computational method that makes use of value judgments, that is, costs of plans and rewards for objectives. Refer to Schut and Wooldridge [96] for a paper and to Schut, Wooldridge and Parsons [98] for an article concerning algorithms that may be employed to control meta-level reasoning.

Bratman [15] explains that there are three kinds of decisions about whether or not to reconsider an intention. (1) "nonreflective" (non-) reconsideration is when no thought (reflection) goes into whether to reconsider; the decision is based on "underlying habits, skills, and dispositions," (2) "deliberative" (non-) reconsideration is when some thought does go into whether to

reconsider, but this happens rarely, says Bratman, and (3) "policy-based" (non-) reconsideration is when the agent, through previous reasoning, has formed a policy about when to reconsider certain intentions, however, this policy may be overridden. Our classification does not match Bratman's.

Lastly, reconsideration can be more or less extensive, affecting plans shallowly or more deeply [15]. For instance, one can either rethink a sub-intention but not the larger intention of which the former is a part, or one may have to rethink the sub-intention because of rethinking the original, larger intention.

2.3.4 Existing Implementations of the BDI Model

First, Section 2.3.4 is a discussion of PRS, the most successful system that can be regarded as a BDI system. Next, Section 2.3.4 shortly mentions a programming language for agent design, consciously built upon the BDI model. Lastly, Section 2.3.4 gives a brief overview of six architectures with some or other component implemented on the BDI framework.

PRS

The *Procedural Reasoning System* (PRS) was formally introduced by Georgeff and Lansky in 1987 [40]. PRS is presented rather as a *system* for controlling mobile robots than a *language* for reasoning about their behavior. Subsequent work has extended the original PRS [39, 47, 46]. Amongst others, PRS has been implemented for malfunction handling for the Reaction Control

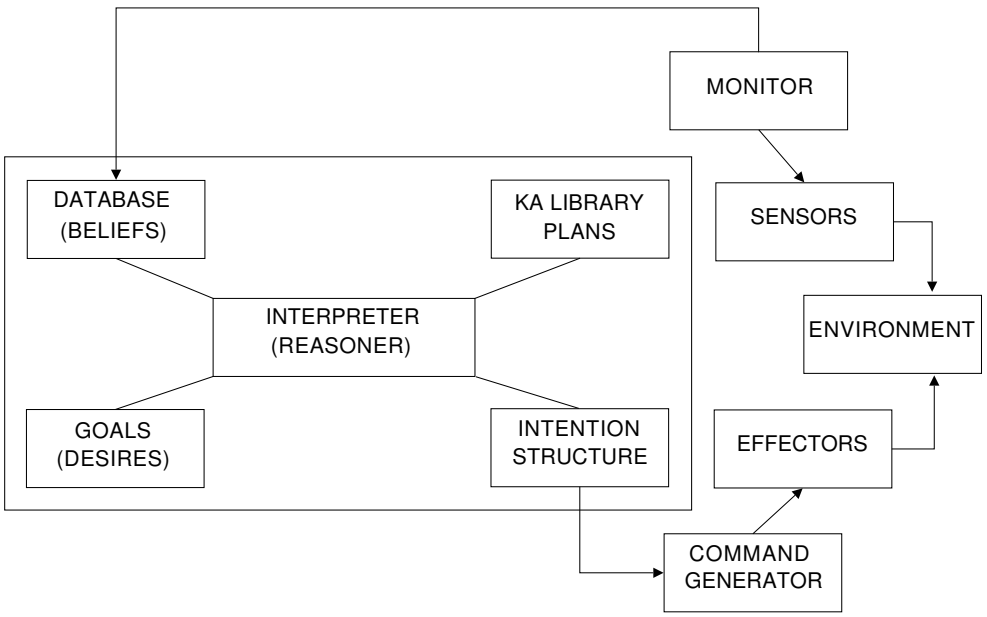

Figure 2.10: PRS system structure.

System of NASA's space shuttle [39]. I shall describe only the basic PRS as presented in [40, 39].

Figure 2.10 represents PRS's system structure [40, 39]. PRS is endowed with the attitudes of belief, desire and intention. It also has a library of plans; procedures, including tests, to deal with various situations. Because the robot or system that PRS is meant to control must operate in uncertain and changing conditions, PRS must be sufficiently reactive and goal-driven, able to interrupt and abandon the *execution* of a plan and the process of *planning*.

> Knowledge about how to accomplish given goals or react to certain situations is represented in PRS by declarative procedure specifications called *Knowledge Areas* (KAs). Each KA consists of a *body*, which describes the steps of the procedure, and an *invocation condition* that specifies under what situations the KA is useful. [40, p. 679]

KAs have (uninstantiated) subgoals and are thus both partial plans and hierarchical. In other words, PRS plans are *schematic* in that they contain subgoals naming other (sub)plans. Some plans consist of primitive actions so that execution of a plan scheme eventually bottoms out in execution of (primitive) actions. Only near-term portions of a plan are executed. Plan selection, formation and execution are interleaved, with beliefs being updated—in the fashion of a production system—whenever an event occurs that may warrant an update. As beliefs change, new intentions may be adopted, which are not due to 'regular' means-ends reasoning.

Goals appear both on the desire/goal stack and as part of the KAs currently in the intention structure. As mentioned earlier, goals in PRS represent desired behaviors (KA procedures) rather than states to be achieved. A goal is the name of a skill, that is, if a goal matches the *invocation condition* of a KA, the KA's *body* is a skill that realizes the goal. For example $walk(a, b)$ ('walk from location a to location b') may be a goal predicate, and some KA should be a hierarchical, partial plan to get an agent from location a to location b.

The intention structure maintains priority over plans (some plans must be executed before others), and some intended plans have conditions of execution (they await certain conditions before being allowed to execute).

The PRS interpreter runs the entire system, manipulating the four components (beliefs, desires, intentions, plans). The interpreter cycles as follows: perceive $\Longrightarrow$ update beliefs $\Longrightarrow$ update active goals $\Longrightarrow$ update intention structure $\Longrightarrow$ select an action to execute $\Longrightarrow$ perceive ... A new intention may become active—if it becomes the KA with highest priority in the structure—else, the KA previously active, remains active.

"The result is focused, goal-directed reasoning in which KAs are expanded in a manner analogous to the execution of subroutines in procedural programming systems," [39, p. 975].

The authors [39] also discuss the importance of commitment, and they discuss commitment to

(i) plans and to (ii) goals.

They distinguish between 'weak' AI planning: selection of a pre-written plan to accomplish a goal, and 'strong' AI planning: generating a sequence of actions by simulating their effects, to achieve a given goal. With a rich set of object-level KAs, 'weak' planning can be effective, and this is the approach that PRS takes.

PRS can be used as a "conditional sequencing system" to implement the sequencing layer of a three-layer architecture [36]. In Gat's words, a "sequencer's job is to select which primitive Behavior (small unit of control [54]) the low-level controller should use at a given time, and to supply parameters to the Behavior," [36, p. 201]. Conditional sequencing is a complex model of plan execution motivated by "human instruction following." It is more complex than the "universal plan" approach, which is a precomputed plan that gives the correct primitive for a robot to use—for a particular task—in each possible state it can be in [36].

Several extensions or dialects of PRS have been developed, for example UM-PRS [60] and PRS-Lite [72].

AgentSpeak

Rao designed an agent programming language called AgentSpeak(L) based on logic programming and BDI theory [83]. The motivation for the development of AgentSpeak(L) was as an abstract language to facilitate the understanding of the relationship between practical implementations of the BDI architecture such as Procedural Reasoning System (PRS) and the formalizing and reasoning proving claims about the behavior of abstract BDI agents. "Sophisticated, multi-modal, temporal, action, and dynamic logics have been used to formalize some of these notions," [83, p. 42]

Since Rao's publication in 1996 there has been much work on extending and further formalizing the language [45] and it has also been implemented as several practical systems [109]. This is probably due to its apparent 'simplicity and elegance' [45] and because of the confidence researchers and programmers might have in the language because of its formal connection with BDI logics.

The various extensions and dialects of the original language have become generally known as AgentSpeak.

Agent Architectures with a BDI Component

There are several agent architectures without a BDI component that employ real-time generative planners. Such architectures are mentioned in Appendix A. In this section, we discuss only

agent architectures that have been developed with a BDI component. In Chapter 3, some BDI architectures with a generative planning component are reviewed.

InteRRaP is a "pragmatic BDI architecture" for agents in multi-agent systems, "where the agent's mental state is distributed over a set of layers," [32]. InteRRaP aims to combine the advantages of layered architectures for intelligent agents, with the advantages of the BDI model. The layers are: a behavior-based layer, a local planning layer and a cooperative planning layer for multi-agent tasks.

JAM is a hybrid intelligent agent architecture that draws upon the theories and ideas of the Procedural Reasoning System (PRS), Structured Circuit Semantics (SCS), and "Act plan interlingua", [44].

> Furthermore, JAM draws upon the implementation pragmatics of the University of Michigan's and SRI International's implementation of PRS (UMPRS and PRS-CL, respectively). JAM provides rich and extensive plan and procedural representations, metalevel and utility-based reasoning over multiple simultaneous goals, and goal-driven and event-driven behavior that are an amalgam of all of the sources listed above. [44, p. 236 (abstract)]

"RETSINA is an open multi-agent system that provides infrastructure for different types of deliberative, goal directed agents. In this sense, the architecture of RETSINA agents exhibits some of the ideas of BDI agents," [75, p. 148].

Saphira is a behavior-based architecture [54]. At the level of control, the problem is decomposed into small units of control called basic behaviors, like obstacle avoidance or corridor following. Behaviors are written and combined using fuzzy-logic techniques.

> Behaviors provide low-level situated control for the physical actions affected by the system. Above that level, there is a need to relate behaviors to specific goals and objectives that the robot should undertake. This management process involves determining when to activate or deactivate behaviors as part of the execution of a task, as well as coordinating them with other activities in the system. [54, p. 229]

PRS-Lite [72], a BDI based system, fills this role of high-level management of goals and behaviors in Saphira.

Burkhard et al. [18] discuss their *Double Pass architecture*. The architecture has options, from which "persistent states for the future", that is, desires and intentions, are derived. It implements goal-directed behavior inspired by the BDI-approach. Their architecture does not employ an extant BDI system in a modular manner; but the architecture has identifiably 'BDI' elements: There is a *deliberator* and an *executor*. The deliberator is not tightly bound by time; it is responsible for more complex, time consuming decision making, and delivers partial hierarchical plans to the executor. New intentions (plans) are being prepared while current

intentions are being executed by the executor. The executor makes just-in-time decisions that are restricted to a minimum and need the most recent sensory information. Intentions are thus involved at both the planning (deliberator) and reactive (executor) levels of the Double Pass architecture.

Kim, Shin and Choi [51] present a plan-based control architecture for intelligent robotic agents with three layers: *deliberative*, *sequencing* and *reactive* layers. The deliberative layer is implemented with UM-PRS [60] as a kind of plan executive employing BDI theory: it is composed of five primary components: a *world model*, a *goal set*, a *plan library*, an *interpreter*, and an *intention structure*.

For completeness, the BDI architectures with generative planning that were reviewed in Chapter 3 are repeated here: Propice-Plan [26], the architecture with Propositional Planning [67], CanPlan [91] and the architecture 'augmented with deliberative planning techniques' [108].

2.3.5 Discussion and Conclusion

In this chapter, the basic background theory required for an understanding of the main contribution of this dissertation have been covered. Only the necessary theory of the respective larger bodies of work in decision theory, logic programming and BDI theory have been covered.

Tambe [38] mentions some short-comings of BDI models that may be addressed by SOAR, such as chunking and a truth maintenance system. And Georgeff mentions the importance of plan caching of generic plans, instead of creating "every new plan from first principals," [38, p. 4].

An advantage that a language like ReadyLog has over BDI models is that ReadyLog has a very tight integration between its planning processes and its 'meta' control. This integration should promote efficiency in performance of ReadyLog systems and easier proofs about the systems. The logic theorem proving approach—of for instance, ReadyLog—is not necessarily at odds with the BDI approach, though. The work in this dissertation hints at the possibility of their marriage, and Sardina and Lespérance have done work to bring Golog and BDI theory together [92].

Something may be said to clarify our understanding of the levels of control in BDI agents: *Object-level* reasoning is reasoning or organizing the 'objects' of practical reasoning—these objects are (mainly) intentions and actions—organizing intentions is deliberation and organizing actions is planning. Hence, object-level reasoning is practical reasoning. Note though, that deliberation *is* a higher level of reasoning than that of planning. *Meta-level* reasoning (or *meta-reasoning* or *meta-level control*) is reasoning about reasoning, that is, thinking about decisions. Reconsideration is thus meta-level reasoning. Also notice that employing a commitment

strategy is meta-level reasoning—but one may feel uneasy saying that a commitment strategy is meta-level *control*: Whereas a reconsideration strategy directly influences whether an agent will deliberate or not, a commitment strategy does not 'control' reasoning as directly. Therefore, we can list the kinds of reasoning under discussion, ordered in ascending levels of control, as: planning, deliberation, commitment-consideration then re-consideration.

The next chapter will set out the development of a logic based POMDP planner. Viewed as a programming language, it is called PODTGolog. This is the planner that is used by the BDI architecture that forms the main contribution of the present work.

Chapter 3

Related Work

In this chapter, we shall review some of the approaches mentioned in the literature for specifying and achieving agent/robot control. We have grouped the related literature into three categories. Section 3.1 deals with logics that allow reasoning over stochastic actions, Section 3.2 deals with Golog dialects that allow reasoning over stochastic actions and Section 3.3 deals with BDI-based architectures and languages with *generative* planning. IndiGolog is an important language with respect to this work that does not fit into any of the three categories. It will be reviewed in the *Discussion* section at the end.

Within the categories, the related works are organized chronologically, from earliest to latest. In Section 3.1, BHL's approach will be given more attention, because it is most relevant to the present work.

3.1 Logics for Dynamical Stochastic Domains

As discussed in Section 2.1, one may include models of probabilistic uncertainty into an agent or system, and to that, one may add notions of utility. Agents modeled with both probabilistic uncertainty and utility, are called *decision-theoretic* agents. Only few *logics* take decision theory into account, that is, few *logics* cater for decision-theoretic agents explicitly. Agents whose stochastic component only are modeled (utility not considered), will be called *stochastic* agents in this chapter. This section covers formal languages for both *decision-theoretic* and *stochastic* kinds of agents.

3.1.1 ICL$_{SC}$

Poole [80] presents a language to combine decision theory and logical representations of actions (based on situation calculus), called the "Independent Choice Logic employing Situation

Calculus to represent change" (ICL_{SC}). He states that ICL_{SC} is for representing MDPs and POMDPs[1], in other words, stochastic action and sensor models can be represented by the language, and utility can also be represented. ICL_{SC} agents are thus decision-theoretic agents. He defines conditional decision theoretic plans in the language, however, he does not provide an algorithm for automatic plan generation. Poole says that ICL_{SC} does not capture the agent's beliefs, but an agent's state is represented sufficiently to program it to do the 'right' thing.

Poole takes a Bayesian decision theoretic approach to agent modeling and planning. He says, "It [ICL_{SC}] is closely related to structured representations of POMDP problems. The hope is that we can form a bridge between work in AI planning and in POMDPs, and use the best features of both," [80, p. 27].

3.1.2 BHL's approach

Bacchus, Halpern and Levesque [2] (BHL) supply a sound theory and specification for reasoning with noisy sensors and graded belief. They provide a way to 'carry along' the graded knowledge of sensor data—making it possible to change it and reason with it at any time in the future. Their whole approach is not formulated as a logic: they use the situation calculus to specify their approach but some elements fall outside the logical language.

Intuitively, their aim is to represent an agent's uncertainty by having a notion of which configuration of situations are currently possible; the *possible-worlds* framework. Then further, each possible world is given a likelihood weight. With these notions in place, they show how an agent can have a belief (a probability) about any sentence in any defined situation. They show that the way beliefs are updated in their approach is equivalent to the standard Bayesian belief update formula. Their work does not, however, cover planning.

$K(s', s)$ is the accessibility relation used in BHL's interpretation of the possible-worlds framework. It is true when situation s' is accessible from situation s. When a situation s' is accessible, it means that, according to the robot's current knowledge, it could possibly be in situation s' without contradicting its own beliefs. Hence, the robot may believe that it is possibly in one of several situations, but due to its incomplete knowledge, the robot does not know which.

$Oi(a, a', s)$ is the *observation-indistinguishability* predicate. It specifies which actions a' are indistinguishable from an action a in situation s. (It is similar to the $choice(a, n, s)$ of DTGolog [13].)

$$K(s'^{+}, do(a, s)) \equiv$$
$$\exists a', s'.K(s', s) \wedge Oi(a, a', s) \wedge s'^{+} = do(a', s') \wedge Poss(a', s').$$

[1] 'Markov decision process' is abbreviated *MDP* and 'partially observable Markov decision process' is abbreviated *POMDP*.

This axiom states that all situations $do(a', s')$ are in the agents knowledge set if and only if it believes it might also be in situation s' (accessible from s) and action a' is indistinguishable from a in s and a' is executable in s' (Poss(a',s')).

$p(s', s)$ is the *relative weight* of the robot's belief that it is in s' and s; that is, when s specifies an action history, s also specifies a set of other states s' that the robot might be in, however, there is a different likelihood of being in any one of s'—the likelihoods are captured by $p(s', s)$. Whereas K tracks what situations are *possible*, BHL define p to track *how* possible a situation is. They introduce a successor-state axiom[2] for p:

$$
\begin{aligned}
&p(s'^{+}, do(a, s)) = \\
&\quad \textbf{if } (\exists a', s', s'^{+}).Oi(a, a', s) \wedge s'^{+} = do(a', s') \wedge Poss(a', s') \\
&\qquad \textbf{then } p(s', s) \times l(a', s') \\
&\qquad \textbf{else } 0.
\end{aligned}
$$

Some important things to note about p are

- $p(s', s)$ is not a probability; it only needs to be greater than zero;
- comparing two accessible situations for likelihood is done by comparing their *relative* $p(s', s)$ weights; that is, p specifies only an ordering of likelihood;
- each time an action a is done, p's successor-state axiom updates the likelihoods (relative weights) of possible worlds using some probability distribution $l(a, s)$, the likelihood of action a in situation s;
- the antecedent of the conditional is similar to the definition of the K successor-state axiom, however, $K(s', s)$ is left out because this constraint is captured by the fact that $p(s', s) = 0$ whenever $K(s', s)$ does not hold; and this effect is propagated by the **else** part of the conditional statement; in other words, $p(s'^{+}, do(a, s)) = 0$ when $K(s', s)$ does not hold, because then $p(s', s) = 0$.

An MDP deals with stochastic decision making but does not facilitate general reasoning about stochastic beliefs. BHL go beyond the MDP model by providing $\textsc{Bel}(\phi, s)$, the agent's (probabilistic) degree of belief in the formula ϕ. It is calculated as the ratio of the sum of those $p(s', s)$ where $p(s', s) \wedge \phi[s']$ is true (while $K(s', s)$ holds), to: the sum of *all* the $p(s', s)$ (while $K(s', s)$ holds). ($\phi[s']$ is the formula ϕ with the situation term s' introduced.):

$$
\textsc{Bel}(\phi[s_{now}], s) = \sum_{\{s' : \phi[s_{now}/s']\}} p(s', s) / \sum_{s'} p(s', s).
$$

[2] In the axiom, **if-then-else** is a macro/abbreviation for a logical sentence.

So BEL is defined in terms of $p(s', s)$. Therefore beliefs are updated whenever $p(s', s)$ is updated, which is whenever an action is performed and p's successor-state axioms is called. Hence, the robot's beliefs change whenever it moves or senses.

Furthermore, "A logical consequence [...] is that BEL(ϕ, s) is a probability distribution over the situations K-related to s," [2, p. 15]. Keep in mind though that the probability distribution is with respect to ϕ, so it should not be confused with probability distributions of *action outcomes* captured by $l(a, s)$. BEL is based on p, and p is the likelihood of being in some situation. These relative weights get their initial values from the robot designer; in the initial situation S_0, all situations K-related to S_0 must be given a weight.

Their approach does not address plans or planning in any way. The present work, in a sense, extends their work with plan generation. A key feature to accomplish plan generation for agents with noisy sensing, will be an update mechanism for probabilities similar to BHL's successor-state axiom for p.

3.1.3 Bonet and Geffner's approach

Bonet and Geffner [10] present a problem solving framework and not purely a logic. In their work, planners are meant to produce "controllers" for dynamical systems (agents). They do not fix on any one planner in their approach. This is possible because they do not specify planning on the semantic level. They investigate mainly dynamic programming planning algorithms, including the "real time dynamic programming" approach to solve POMDP and nondeterministic problems. Their framework can be used to design stochastic and decision-theoretic agents.

Bonet and Geffner [10] use a "logical representation language" [37] to represent mathematical models of the problem domains. The language is an extension of the class of STRIPS-style languages for representing planning problems. It can specify fluent predicates, action preconditions, action effects, state transitions, probabilities of stochastic outcomes, observability, sensor feedback and probabilistic models of sensors. In general, the language can represent state models and action/sensor dynamics (for POMDPs, for example) and it incorporates various features for facilitating probabilistic planning with incomplete information (for example, temporal extensions of Bayesian networks).

What makes Bonet and Geffner's approach different from ours, is firstly, that they intend their framework to be able to solve a variety of AI planning problems, whereas this dissertation concerns plan generation for *autonomous agents* specifically, and secondly, we fix on a *POMDP* solver specifically.

3.1.4 $\mathcal{ESP}$

A new logic for reasoning about stochastic action and noisy sensing is $\mathcal{ESP}$ [35] ("$\mathcal{ESP}$ = $\mathcal{ES}$ + uncertainty"). That is, it facilitates stochastic agent design. It supersedes $\mathcal{ES}$ [61] and $\mathcal{AOL}$ [59]. Gabaldon and Lakemeyer [35] derive some of their inspiration from Bacchus, Halpern and Levesque [2].

$\mathcal{ESP}$ provides a knowledge operator K, and belief operator $HasP(\alpha, p)$ meaning: statement α has probability p (similar to BHL's $\textsc{Bel}(\phi, s)$). $\mathcal{ESP}$ is a 'situation' based logic, but does not include situation terms. Also, unlike BHL's $\textsc{Bel}(\phi, s)$, $HasP(\alpha, p)$ has a syntax and semantics defined in the language and logic. Generative planning is not dealt with in $\mathcal{ESP}$.

Just as in all the literature mentioned so far in this section, in $\mathcal{ESP}$ too, one can model the dynamics (including sensing) of agents in stochastic domains.

3.1.5 DyMoDeL

Finally (for this section) DyMoDeL[3] [87] is a logic in development which has the same applications as $\mathcal{ESP}$.

Whereas $\mathcal{ESP}$ has a modal 'flavor', DyMoDeL is explicitly a modal logic. In DyMoDeL, a formula like $\{o|a\}\mathbf{B}_{=0.5}\varphi$ means 'After observing o given a was executed, the belief in φ will be 50%'.

With the current version, agents modeled as POMDPs can be designed, but DyMoDeL does not provide a procedure for automated policy generation.

3.2 Golog Dialects for Stochastic Domains

Although the DTGolog language is a Golog dialect for stochastic domains, it is discussed in Chapter 2. The reader may want to consult Section 2.1 before or while reading this section.

3.2.1 stGolog

In his monograph, Reiter [84] describes how to implement an MDP and a POMDP system. He defines the language stGolog, standing for *stochastic Golog*. Although, in a different chapter, Reiter fully covers the *accessibility relation* to accommodate the representation of an agent's knowledge, in his chapter on representing (PO)MDPs, he opts not to use the accessibility relation. Also, he does not define an explicit *belief* state; he captures the agent's belief only via the

[3]The developers of DyMoDeL have considered changing the name of the logic to POMDL.

probabilities involved with stochastic actions and stochastic observations. Lastly, Reiter does not provide a method to automatically generate (optimal) policies, given a decision domain theory; he only provides the tools for the designer to program policies for partially observable decision domains by hand.

As far as sensing in stGolog is concerned, Reiter [84] associates a sense action with each (normal) stochastic action. Hence, in a policy, after the agent performs an action, the action's sensor is activated, which returns an observation. Non-sensing actions and sensing actions are thus differentiated. In *our* semantics of a POMDP, we do not make this distinction; actions can be for sensing or not. We associate each action directly with a set of observations for that action. That is, each action in our semantics has a sensing component. This is similar to Bacchus, Halpern and Levesque [2] and to Kaelbling, Littman and Cassandra [48].

However, we take a hint from Reiter [84] in specifying sense outcome conditions: he has a fluent $outcomeIs(o, s)$ where s is a situation term and o is "the outcome chosen by nature for the immediately preceding stochastic sense action," [84, p. 367]. Hence, conditional plans can branch on whether $outcomeIs(o, s)$ holds for various outcomes o of some sensing action. In our semantics there is an $ObservationIs(o)$ predicate, where o is an *observation* term, a kind of term stGolog does not have. However, $ObservationIs(o)$ serves the same purpose as $outcomeIs(o, s)$; they both mark decision points in a conditional plan. A definition of $ObservationIs(\cdot)$ will be given and discussed in detail in Chapter 4.

3.2.2 pGolog

In Chapter 6 of Grosskreutz's Ph.D. dissertation [42], he shows how the Golog framework "can be extended to allow the projection of high-level plans interacting with noisy low-level processes, based on a probabilistic characterization of the robot's beliefs," [p. 95]. Low-level processes with uncertain outcomes are modeled with probabilistic programs. He calls the extension to Golog *pGolog*. According to Grosskreutz, the intuition behind pGolog is that a program's execution can be simulated, and the probabilities of the various possible outcomes of the program noted. "As a result of this probabilistic setting, projection now yields the probability of a plan to achieve a goal, which leads us to the notion of probabilistic projection," [42, p. 95].

Grosskreutz follows BHL [2] in the representation of an agent's uncertainty about the state of the world. That is, he characterizes the agent's beliefs by a distribution over possible situations considered possible.

In his dissertation, Grosskreutz defines $prob(p, \sigma_1, \sigma_2)$, the probabilistic branching instruction, where σ_1 and σ_2 are pGolog programs and p is a probability which lies between 0 and 1, exclusive. "The intended meaning of $prob(p, \sigma_1, \sigma_2)$ is to execute program σ_1 with probability p, and σ_2 with probability $1 - p$," [42, p. 97].

Later in his Chapter 6, he shows how probabilistic projection in pGolog can be applied to expected utility, which leads to decision theory.

Whereas Chapter 6 of his dissertation deals with a robot's projections with probabilistic uncertainty, his Chapter 7 is about belief update of the robot's epistemic state. Grosskreutz explains the difference between probabilistic projection and belief update as (in the former case) asserting the probability of a situation according to how the world might evolve, and (in the latter case) changing the agent's beliefs (probability of current situation) according to actual actions (recently) performed. He makes clear the distinction between the tasks of probabilistic projection and belief update, saying that the former does not involve any actual execution or sensing information, while the latter deals with actual execution and sensing information. pGolog concerns high-level programs "that appeal to the agent's real-valued beliefs at execution time." In addition, belief update in pGolog does not only update beliefs, but also the state of the progress of the current program.

Grosskreutz does not employ (PO)MDPs in pGolog. Instead, he does probabilistic projection of specific programs. He does however make use of expected utility to decide between which of several programs to execute (after simulated scenarios). He does not make use of an epistemic accessibility fluent like BHL [2] or $\mathcal{ESP}$ [35]. In order to capture the idea of accessibility, he sets the probability of program transitions to 0 if s' is not accessible from s_{now}, and p otherwise. (See [42, p. 139] concerning *transition semantics*).

As with stGolog, pGolog does not allow for plan generation, but it does allow for plan specification.

3.2.3 POGTGolog

Finzi and Lukasiewicz [30] present a game-theoretic version of DTGolog to operate in partially observable domains. They call this extension *POGTGolog*. This is the only Golog dialect that can take partially observable problems as input, that is, that has some kind of POMDP solver for agent action planning.

Developers who prefer a Golog dialect for agent programming but desire their robots or agents to operate with POMDP information cannot easily modify POGTGolog to work with single robots. The present work is not merely a simplification of Finzi and Lukasiewicz's; rather, it extends DTGolog and uses several elements in POGTGolog—either directly or for inspiration.

One major adaptation that Finzi and Lukasiewicz [30] make to DTGolog is their replacing of the situation term in the argument list of the *BestDo* formula with a *belief state* term b. b is a set of pairs (s', p) where s' is a situation believed possible with probability p. This change influences all other semantics that would have taken the situation term as argument, having the

effect of POGTGolog operating over belief states instead of situations.

Our approach is to clearly divide specifying probabilities for actions $probNat(\cdot)$ and for observations $probObs(\cdot)$ (see details in Section 4.1), whereas Finzi and Lukasiewicz [30] combine these probabilities by defining $prob(a, n, s, o)$ to be the probability of observing o in situation s, given action a and one of its realization n in the environment. In fact, $prob(a, n, s, o) = probNat(n, a, s) \times probObs(o, a, s)$.

Much of their work concerns the details of specifying the semantics and syntax for a (multiagent) game theoretic situation calculus [30]. We sought only the elements useful for a *single* agent specification.

3.2.4 ReadyLog

Ferrein and Lakemeyer [28] present an agent programming language (APL) called ReadyLog. Approximately ten years after Golog's birth, ReadyLog combines many of the disparate useful features of various dialects into one package. Some of the capabilities of ReadyLog are:

- to reason about and manipulate programs in the language of ReadyLog; the programs are turned into objects, that is, they are reified;
- sensing is incorporated: reasoning about actions that are not meant to have an effect on the world, but only for information gathering;
- decision-theoretic (probabilistic) planning is catered for;
- plans can be incrementally executed, that is, the interleaving of acting and planning, resulting in online planning;
- probabilistic projections can be done;
- continuous passive sensing can be done (versus sensing only as planned actions);
- monitoring execution of policies to check their validity and the need for re-planning; and
- significantly faster planning due to enhancements in domain modelling.

Table 3.1 shows which features various Golog languages have ('+') and do not have ('-'). Note that only DTGolog and ReadyLog incorporate decision theory; this decision theory is however for fully observable Markov decisions only.

Because ReadyLog is a Golog dialect, plan specification is possible. Moreover, ReadyLog is capable of plan *generation* due to it subsuming DTGolog.

	Online	Sensing	Exog. Actions	Conc. Exec.	Projec-tion	Prob. Actions	Cont. Change	Decision Theory
Golog	-	-	-	-	-	-	-	-
ConGolog	-	-	-	+	-	-	-	-
IndiGolog	+	+	+	+	+	-	-	-
ccGolog	+	+	+	+	+	-	+	-
pGolog	+	+	+	-	+	+	-	-
DTGolog	-	-	-	-	+	+	-	+
ReadyLog	+	+	+	+	+	+	+	+

Table 3.1: Features of Golog Languages (Tbl. 4.1 in Ferrein's PhD dissertation [29]).

Note in particular, from Table 3.1, that ReadyLog incorporated (the capabilities of) pGolog [42] and (offline) DTGolog [13]. ReadyLog thus provides the capability to write formulas involving *belief*, via the BEL function provided by pGolog [42] and BHL [2]. And ReadyLog can also solve MDP problems, generating conditional policies.

Ferrein and Lakemeyer mention in their article [28], how software agents programmed in ReadyLog perform. These agents fare reasonably against opponents in a particular software environment. The authors also mention ReadyLog-controlled robots in the service/personal robotics domain and in teams of soccer robots. The robots also fare competently in last mentioned domains.

3.3 BDI-based Architectures with Generative Planning

This section reviews the literature that reports specifically on extending the belief-desire-intention (BDI) model of agency with a form of generative planning. In a sense, this is exactly what the research in this dissertation is about. The differences in the different research products of this section is in the details of how the extension is accomplished, its motivation, and its application. The reader may want to consult Section 2.3 before or while reading this section.

3.3.1 Propice-Plan

Propice-Plan [26] is a language for implementing the control of dynamic systems in real-world situations. Despouys and Ingrand [26] present a unified approach for planning and execution, combining plan synthesis and anticipation planning and a BDI-based language called Propice for supervision and execution control.

The authors alternate between calling their approach a *language* and a *framework*[4].

[4]When it comes to (sophisticated) programming languages for agents, the distinction between the specification of the language and an architecture (framework) seems to be less clear.

They "use the term OP (operational plan) as the result of a planning activity, but also an operational procedure defined by a domain expert," [26, p. 279]. When a goal needs to be achieved and none of the available OPs is applicable to achieve the goal, a planner is called to *synthesize* information in the available OPs to produce a new OP. They also introduce a novel approach to *anticipate* the best OP to execute at some future point: They highlight that the Propice component of the framework does not generate plans, it only selects and supervises plans. "Our goal is to take advantage of the large number of OPs available and of the spare time left while performing control execution and supervision. The whole idea of anticipation planning is thus based on the simulation of OPs executive information," [26, p. 286].

Propice-Plan is a much more sophisticated framework for management of goals than the architecture presented in this dissertation (cf. BDI-POP, Chapter 5). Nevertheless, plans (OPs) in Propice-Plan cannot take advantage of stochastic models of a system's description.

3.3.2 Propositional Planning in BDI Agents

Meneguzzi, Zorzo and Da Costa Móra [67] note the inefficiency of traditional plan generation in real-time, compared to the use of (efficient) pre-compiled plans. But they also note the disadvantage of the inflexibility of pre-compiled plans compared to the relevance of real-time generated plans, and the need for the agent designer to build the plans before the deployment of the agent. Hence, they describe "the relationship between propositional planning algorithms and means-end reasoning in BDI agents," [67, 63] and define a mapping "between the structural components of a BDI agent and propositional planning problems," [67, 63]. This mapping then allows state of the art planners that conform to the mapping to be utilized in their approach, while retaining the advantages of BDI-style practical reasoning. The authors state that the main contribution of their work consists in the definition of a mapping from BDI means-end reasoning to fast planning algorithms.

They [67] take a relatively sophisticated BDI model (X-BDI) and modify it for their purposes. To verify their approach, Meneguzzi, Zorzo and Da Costa Móra implement the propositional planning algorithm GRAPHPLAN in their prototype. The implemented planner acts as the means-end reasoner in the BDI model. GRAPHPLAN does not deal with stochastic domain models.

3.3.3 CANPLAN

Sardina, De Silva and Padgham [91] present CANPLAN, a BDI language based on CAN and AGENTSPEAK. CANPLAN includes actions with preconditions and effects, multiple variable bindings, and an account of declarative goals.

"A central distinguishing feature of CAN is its Goal(ϕ_s, P, ϕ_f) goal construct, which provides

a mechanism for representing both declarative and procedural aspects of goals. Intuitively, a *goal-program* Goal(ϕ_s, P, ϕ_f) states that we should achieve the (declarative) goal ϕ_s by using (procedural) program P; failing if ϕ_f becomes true," [91, p. 1003].

"By building on the underlying similarities between BDI systems and Hierarchical Task Network (HTN) planners, we present a formal semantics for a BDI agent programming language which cleanly incorporates HTN-style planning as a built-in feature," [91, p. 1001 (abstract)]. Through their semantics, they are implicitly also specifying an architecture.

The authors specify both the HTN planner and the BDI interpreter in a consistent semantics, thus integrating the formal operational semantics of their BDI system and HTN planner. They purport that no other language integrates generative planning with a BDI model with a *formal* semantics.

CanPlan also includes the *Plan*(P) construct: *Plan* instantiates HTN planning. It searches offline for a complete hierarchical decomposition (partially ordered sequence of primitive actions) of program P using HTN-style reduction of a compound task into actions. *Plan* forms part of BDI style programs as specified by the program designer / domain expert at specific points in the program. It provides programmer control over when to plan. Note that HTN planning and POMDP planning are quite different, and have different pros and cons.

Sardina, De Silva and Padgham mention that their work [91] is possibly most related to the IndiGolog language of De Giacomo and Levesque [23], which is reviewed later in this section.

3.3.4 Augmenting BDI Agents with Deliberative Planning Techniques

Walczak, *et al.* state

> Due to advances in planning techniques and understanding of *planning problems*, it seems reasonable and interesting to combine the strength of flexible means-end reasoning given by deliberative [generative] planners with the timely reactivity and goal deliberation capabilities carried [out] by BDI systems. [108, pp. 113–114]

In their work, "augment the BDI system with a relatively simple planner that is invoked from the BDI controller and used for the purpose of creating short-term plans that need a proof of correctness," [108, p. 114]. This sounds similar to the work of this dissertation, however, in this work, plans do not "need a proof of correctness" because they are logically sound by definition (cf. Chapter 4).

Their planner employs a state-based search algorithm working on an agenda of states. States are objects including components such as: sets of objects with value-assigned attributes, actions, a stack of goals weighted with utility measures and a transition function. The utility

functions that occur in the planner are interpreted as agent desires. So their work is similar to this dissertation's in that their agents interpret utilities as desires. The planner developed here, has a different solving algorithm to theirs though; the new planner solves specifically POMDP problems (cf. Section 2.1 and Section 4.1).

3.4 Discussion

The agent architecture developed in this dissertation will allow for plans to be generated for decision-theoretic agents. That is why the literature in Section 3.1 is relevant. Further, the POMDP planner/solver we develop will turn out to be a new Golog dialect, and this planner will be called into action in a BDI framework to fulfill means-end reasoning. That is why the literature in Sections 3.2 and 3.3 respectively, is relevant.

IndiGolog [23, 90] is not concerned with stochastic domain models and thus does not fit into Sections 3.2. Because IndiGolog is a Golog dialect, it also employs programs to guide, inform and constrain actions towards a goal. The contribution of the IndiGolog dialect, however, is that the programs it allows may include parts that are executed *online*. The programs of all previous Golog dialects are executed *offline*.

De Giacomo and Levesque [23] explain that offline execution of a program means that a (logically) legal execution of a sequence of actions is sought according to the constraints and tests of the program, for the whole program, before even one action is executed. This is clearly detrimental to the reactivity of the agent being controlled. Furthermore, when tests later in a program depend on information meant to be collected by sensing actions earlier in the program, the sensing actions need to be executed in the world before the test point in the program is reached. In online execution of a program, as choice points are reached (between two or more actions) one action must be chosen and executed immediately in the world; and there is no backtracking in the real world (assuming executions cannot be undone). In offline execution, backtracking is possible when a certain choice of actions leads to failure.

There may be circumstances when the agent programmer prefers portions of a program to be online and other portions offline. IndiGolog allows both kinds of programming to be mixed, by providing the Σ operator to indicate that search is required. For example, given that δ_1, δ_2 and δ_3 are 'regular' Golog programs, $\delta_1; \Sigma\delta_2; \delta_3$ is a program in IndiGolog that will execute δ_1 online, then search for a "globally successful termination" of δ_2, then continue "boldly" with the δ_3 portion of the program.

"The technical machinery [they] use to define on-line program execution in the presence of sensing is essentially that of (De Giacomo, Lespérance, & Levesque 1997 [ConGolog])," [23, p. 29] through the use of the predicates *Trans* and *Final* to define a single-step semantics for

programs. However some adaptation to ConGolog was necessary.

"One of the main advantages of a high-level agent language containing nondeterminism is that it allows limited versions of (runtime) planning to be included within a program. Indeed, a simple planner can be written directly:

$$\textbf{while } \neg\phi \textbf{ do } \pi a.\ (\mathit{Acceptable}(a)?\ ;\ a) \textbf{ endWhile}.$$

[...] this program says to repeatedly perform some nondeterministically selected action [a] until condition ϕ holds[5]," [23, p. 33–34]. In IndiGolog this "planner" loop would be the operand of the Σ operator.

IndiGolog and CanPlan [91] are both similar to the work of this dissertation in that they supply agent designers with the tools to interleave planning and execution. Besides these languages not dealing with stochastic models, there are two main differences between their work and the proposed architecture: (1) They provide formal semantics for their languages, whereas some parts of the proposed architecture are not as rigorously formalized. (2) Their languages allow for planning points to be specified by the programmer, whereas the architecture developed in this dissertation prescribes planning at (conditionally) fixed intervals. This may be a drawback because it takes control away from the programmer, but it could be a benefit when one considers that the programmer need not be concerned with when to plan; the decision of when to plan can be handled intelligently and autonomously.

Not one of the references in this chapter is a combination of a BDI architecture employing a generative planner for *partially observable stochastic* domains.

[5] $\mathit{Acceptable}(a)?$ is a sort of filter with benefits in this context; refer to De Giacomo and Levesque [23].

Chapter 4

Extending DTGolog to deal with POMDPs

If a robot or agent can perceive every necessary detail of its environment, its model is said to be *fully observable*. In many practical applications this assumption is good enough for the agent to fulfill its tasks; it is nevertheless unrealistic. A more accurate model is a *partially observable model*. The agent takes into account that its sensors are imperfect, and that it does not know every detail of the world. That is, the agent can incorporate the probabilities of errors associated with its sensors, and other uncertainties inherent in perception in the real world, for example, obscured objects. If an agent or robot cannot represent the uncertainties inherent in perception, it has to *assume* perfect perception; this assumption either might lead to spurious conclusions or the necessity for additional methods that keep the agent's reasoning reasonable. For sophisticated robots or agents, it may be best to accept and reason with noisy sensor data. One model for reasoning under uncertainty with partial observability is the *partially observable Markov decision process* (POMDP).

This chapter describes the extension of Rens, Ferrein and Van der Poel [85] to DTGolog. They [85] extend the approach of DTGolog [13] in such a way that it can also deal with partially observable domains. An advantage of using a dialect of Golog for the implementation for the planner, is that the integration of beliefs into the situation calculus has previously been done [2, 30] and that work can be used for formulating POMDPs. Further, given a background action theory, an initial state and a goal state (or 'reward function' in POMDPs), Golog programs essentially constrain and specify the search space of available actions. The resulting plan (*policy* in POMDP terms) is a Golog program which can be executed directly by an agent.

We shall call the new dialect PODTGolog for ease of reference. For decision theoretic planning, the *Do* formula becomes *BestDo* of DTGolog, employing the *decision tree rollback procedure* described in Section 2.1. In particular, instead of using *BestDo*, Rens, Ferrein and Van der Poel introduce a predicate *BestDoPO* to operate on a *belief* state (cf. Section 2.1.2) rather than on a *world* state. Whereas DTGolog models MDPs, PODTGolog models *belief-MDPs*. A belief-MDP is one perspective of POMDPs, where the states that are being reasoned over are *belief*

states and not *world* states.

4.1 Semantics of POMDPs in Golog

The signature of the *BestDoPO* formula is

BestDoPO(
program : δ,
belief–state : b,
horizon : h,
policy : π,
policy–value : v,
program–probability : pr
).

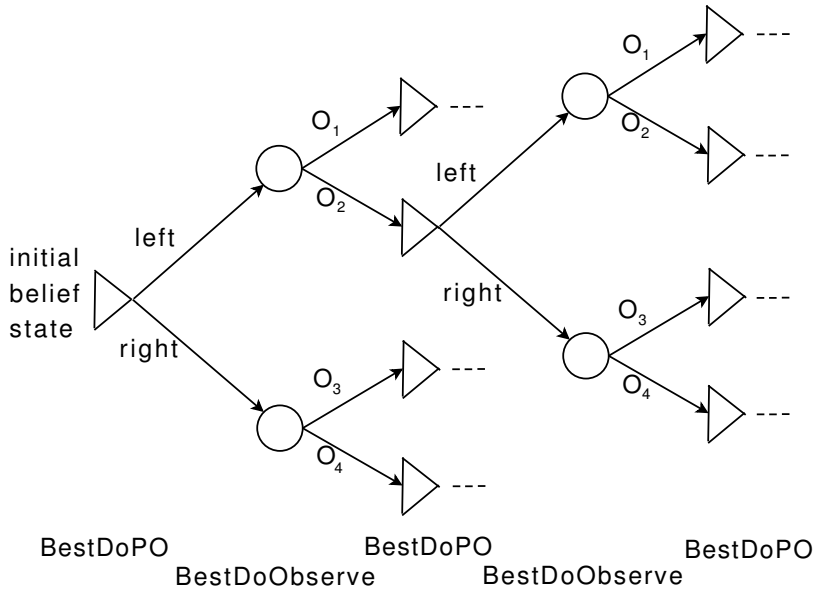

Figure 4.1: *BestDoPO* represented as a belief decision tree.

BestDo solves completely observable Markov decision processes (completely observable MDPs), whereas *BestDoPO* solves 'belief-MDPs', that is, POMDPs. *BestDoPO* makes use of one auxiliary procedure: *BestDoObserve*: it processes the step where nature determines which observations are possible, given the agent's preferred action. The relation of *BestDoPO* to a belief decision tree can be seen in Figure 4.1. Each node (triangle or circle) is a call to one procedure (*BestDoPO* or *BestDoObserve*).

BestDoPO is called with a program (complex action) δ that specifies the basic behavior ex-

pected of the agent, including the actions it may choose from. Initially a belief state b is provided; subsequently, recursive calls are supplied with a 'current' belief state, given a particular action-observation history. The user must provide a planning horizon h; the number of actions for which a policy is required. If the call to *BestDoPO* succeeds, π is instantiated with a policy (conditional plan), v is instantiated with the total expected reward the agent will get if it follows the policy, and the probability that the program will be executed successfully instantiates pr.

An example of how *BestDoPO* may be called initially—with a program that allows the agent to choose between three actions a_1, a_2, a_3 (without constraints), with the initial belief state b_0 and with the user or agent requiring advice for a sequence of seven actions—is

$$BestDoPO(\textbf{while } true \textbf{ do } [a_1 \mid a_2 \mid a_3], b_0, 7, \pi, v, pr).$$

4.1.1 Basic Definitions and Concepts

A belief state b contains the elements (s, p)—a possible (situation calculus) situation s together with probability p (as in Finzi and Lukasiewicz [30]). Note that the belief state b replaces the situation term s in the argument list of *BestDoPO*.

We do not provide a semantics for deterministic actions; such actions can be specified in the domain description by defining a single outcome (nature's choice) with probability 1 for the deterministic action. Also, be aware that locomotive actions are only implicitly distinguished from informational actions by how the designer specifies their realizations by nature (in the environment) and the probability of such a realization occurring. For example, a 'pure' locomotive action has only the null or empty observation which will occur with probability 1, and a pure sensing action can be performed in only one way (with possibly multiple resulting observations)—the intended sensing action occurring with probability 1. In other words, each and every action has a sensing component.

We use the idea of Finzi and Lukasiewicz [30] and assume that an action is possible in a belief state when it is possible in any situation which is part of the belief state, that is, $Poss(a, b)$ iff $\exists s.Poss(a, s)$ where $(s, p) \in b$.

The semantics will treat action terms and observation terms as different sorts. In the definition of *BestDoPO* (all the formulae to follow), when an action is mentioned, it is written without arguments; arguments are being suppressed for simplicity and because they are not required for the definition to be correct. Observation terms are never arguments to action terms.

The predicate $PossObs(o, a, b)$ is added to the action theory, which specifies when an observation o is possible (perceivable) in belief state b, given an action a. The meaning of $PossObs(o, b)$ is unclear—can one ask whether it is possible to *have* an observation? One should rather ask whether it is possible to *make* an observation—and *making* an observation involves the

act of sensing. To clearly distinguish between preconditions for observations and for actions, $Poss(a, s)$ will be referred to as $PossAct(a, s)$.

It is important to note that the b in $PossObs(o, a, b)$ is the belief state reached *after* action a was executed. That is, if a was executed in b' and b is the new state reached, then $PossObs(o, a, b)$ says whether it is possible to observe o *after* a has been executed.

Boutilier *et al.* [13] defined a predicate $prob(n, p, s)$; we define a term and call it *probNat*. $probNat(n, a, s)$ has the same purpose as $prob(n, p, s)$, but $probNat(\cdot)$ is defined differently. They [13] relate the outcome n implicitly to some intended action a, whereas in PODTGolog, the relation needs to be made explicit by giving a as an argument. The definition for PODTGolog is a function (term) which 'returns' a probability, whereas Boutilier *et al.* have their term p unify with the applicable probability. The state transition function $\mathcal{T}$ of Section 2.1 is equivalent to *probNat* in the following way:

$$\mathcal{T}(s, a, s') = p \equiv p = probNat(n, a, s),$$

when all 'resulting' states s' result from executing outcomes n of a, where s is the state in which the agent performs a.

Next, the function $probObs(o, a, s)$ is introduced; it is the probability that o will be observed in s after a was executed in some other situation to reach s. That is, $probObs(o, a, s) = probObs(o, a, do(a, s^-))$, where $s = do(a, s^-)$. And

$$probObs(o, a, s) = O(s, a, o);$$

O is the observation function of Section 2.1.

$ChoiceNat(n, a, s)$ holds when n is an outcome associated with a in s.
$ChoiceObs(o, a, s)$ holds when o is an observation associated with a when executed in s.

We define *probNebs*, which is similar to Equation 2.9 in Section 2.1.2; the probability of being in the new belief state b'. The definition is as follows:

$$probNebs(o, a, b) \doteq \sum_{(s,p)\in b} p \sum_{ChoiceNat(n,a,s),PossAct(n,s)} probNat(n, a, s) \cdot probObs(o, a, do(n, s)).$$

$r(\cdot)$ is the reward function over belief states:

$$r(b) \doteq \sum_{(s,p)\in b} reward(s) \cdot p,$$

where $reward(\cdot)$ is the reward function over *situations*.

Like Finzi and Lukasiewicz [30] and Grosskreutz [42], we too do not use an accessibility re-

lation to reason with beliefs. Although certain properties of the agent's belief system can be guaranteed by the use of an accessibility relation, its use does complicate matters and we have opted to leave it out of our semantics.

4.1.2 The Partially Observable *BestDo*

The semantics of PODTGolog is the definition of *BestDoPO* (partially observable *BestDo*) via several clauses. *BestDoPO* implements Equation 2.10 of Section 2.1.2: $\pi^* = \arg\max_\pi(Vb_{\pi,h}(b_0))$.

1. **Zero horizon**

   When no more actions have to be planned for, the planning horizon has reached zero, that is $h = 0$. There will be no further recursive calls.

$$BestDoPO(\delta, b, h, \pi, v, pr) \doteq$$
$$h = 0 \wedge \pi = Stop \wedge v = r(b) \wedge pr = 1.$$

2. **Null program**

   When all actions have been 'performed', there are no remaining actions in the input program, that is, *BestDoPO* is called on the empty program. There will be no further recursive calls.

$$BestDoPO(nil, b, h, \pi, v, pr) \doteq$$
$$\pi = Stop \wedge v = r(b) \wedge pr = 1.$$

3. **Empty belief state**

   It may happen that the agent's belief state becomes empty. Although the agent designer should make specifications that avoid the agent becoming 'unconscious' (that is, having no beliefs), *BestDoPO* still caters for this case.

$$BestDoPO(\delta, b, h, \pi, v, pr) \doteq$$
$$b = \{\} \wedge \pi = Stop \wedge v = 0 \wedge pr = 1.$$

   There will be no further recursive calls.

4. **Test action**

   Our definition for the test action is adapted from that of Finzi and Lukasiewicz [30]:

$$\begin{aligned} &BestDoPO(\phi?; rest, b, h, \pi, v, pr) \doteq \\ &\quad \exists pr'.\phi[b] \wedge BestDoPO(rest, b, h, \pi, v, pr') \wedge pr = pr' \cdot prob(\phi[b]) \vee \\ &\quad \neg\phi[b] \wedge \pi = Stop \wedge v = 0 \wedge pr = 0. \end{aligned}$$

   Probabilities involved in this formula are influenced in proportion to the agent's *degree of belief* [2]. In the following sentence, $\phi[s]$ is true if and only if ϕ holds in situation s. $\phi[b]$, as it appears in this clause, is true if and only if $\phi[s]$ is true for some $(s, p) \in b$, and it is associated with the probability $prob(\phi[b]) = (\sum_{(s,p)\in b:\phi[s]} p)$, which affects the overall program success probability.

5. **Conditional statement**

$$\begin{aligned} &BestDoPO([\textbf{if } \varphi \textbf{ then } p_1 \textbf{ else } p_2 \textbf{ endif}]; rest, b, h, \pi, v, pr) \doteq \\ &\quad BestDoPO([[\varphi?; p_1] \mid [\neg\varphi?; p_2]]; rest, b, h, \pi, v, pr). \end{aligned}$$

   The "nondeterministic choice of actions" version of *BestDoPO* (see below) is called; however, both 'branches' cannot survive; exactly one must survive: If φ holds, the 'then' branch will execute and the 'else' branch will not because $\neg\varphi$ will not hold. If φ does not hold, the situation is reversed. The meaning of the conditional statement is thus maintained.

   Observe that each 'branch' will call the 'test action' formula, and probabilities will be dealt with according to that formula.

6. **Conditional iteration**

   For completeness, we provide a procedural definition (based on the implementation) of a **while** loop. A formal definition is not provided by Soutchanski [100], and adapting the one by, for example Brachman and Levesque [14], is beyond the scope of this work.

$$\begin{aligned} &BestDoPO([\textbf{while } \varphi \textbf{ do } p]; rest, b, h, \pi, v, pr) \doteq \\ &\quad \neg\varphi[b] \wedge BestDoPO(rest, b, h, \pi, v, pr) \vee \\ &\quad \exists pr'.\varphi[b] \wedge BestDoPO(p; [\textbf{while } \varphi \textbf{ do } p]; rest, b, h, \pi, v, pr') \wedge \\ &\quad pr = pr' \cdot prob(\phi[b]). \end{aligned}$$

   Note that *horizon* has precedence over *iteration*, that is, even if the iteration condition (φ) is satisfied, if h is zero, then planning stops.

7. **Sequential composition**

   This definition decomposes (complex) sequential actions, processes the first action in a composed sequence, and then the rest.

$$BestDoPO([\delta_1;\delta_2];\delta_3,b,h,\pi,v,pr) \doteq$$
$$BestDoPO(\delta_1;[\delta_2;\delta_3],b,h,\pi,v,pr).$$

8. **Nondeterministic choice of actions**

   Out of two possible actions δ_1 and δ_2, the policy associated with the action that produces the greater value (current sum of rewards) is preferred and thus included in the determination of the final policy π. This formula is the heart of the expected value maximization of decision theory.

$$\begin{aligned}
&BestDoPO([\delta_1|\delta_2];rest,b,h,\pi,v,pr) \doteq \\
&\quad \exists\pi_1,v_1,pr_1.BestDoPO(\delta_1;rest,b,h,\pi_1,v_1,pr_1) \wedge \\
&\quad \exists\pi_2,v_2,pr_2.BestDoPO(\delta_2;rest,b,h,\pi_2,v_2,pr_2) \wedge \\
&\quad ((v_1,\delta_1) \geq (v_2,\delta_2) \wedge \pi = \pi_1 \wedge v = v_1 \wedge pr = pr_1)) \vee \\
&\quad ((v_1,\delta_1) < (v_2,\delta_2) \wedge \pi = \pi_2 \wedge v = v_2 \wedge pr = pr_2)).
\end{aligned}$$

9. **Nondeterministic finite choice of arguments**

   DTGolog defines a clause of *BestDo* for the nondeterministic finite choice of arguments defined for Golog: $\wp x.(a_1)$ (nondeterministically choose a value for x in a_1 from a finite, non-empty range of values). PODTGolog's definition is exactly the same as that of DTGolog, except that in PODTGolog, the belief state term b is substituted for the situation term s:

$$BestDoPO([\wp[x:\tau]\delta];\delta',b,h,\pi,v,pr) \doteq$$
$$BestDoPO([\delta|^x_{c_1} \ldots \delta|^x_{c_n}];\delta',b,h,\pi,v,pr).$$

   This clause of *BestDoPO* allows for choices over action arguments: In the program δ, all free variables x are substituted by an element from τ . The domain of $\tau = \{c_1,\ldots,c_n\}$ must be finite. Note that the construct $[\wp[x:\tau]\delta]$ is an abbreviation for $[\delta|^x_{c_1} \ldots \delta|^x_{c_n}]$, where $\delta|^x_c$ means 'substitute c for all free occurrences of x in δ' [100].

10. **Probabilistic observation**

    Here we branch on all possible observations, given the robot's intended action a. $choiceObs'(a)$ is defined similarly to $choice'(a)$ in the discussion of DTGolog in Section 2.2.3. $choiceObs'(a)$

'returns' the set of observations that the robot may perceive:

$$choiceObs'(a) = \{o \mid ChoiceObs(o, a, s) \text{ for all situations } s\}.$$

$$\begin{aligned}
&BestDoPO(a; rest, b, h, \pi, v, pr) \doteq \\
&\quad \neg PossAct(a, b) \wedge \pi = Stop \wedge v = 0 \wedge pr = 0 \vee \\
&\quad PossAct(a, b) \wedge \\
&\quad \exists \pi', v'.BestDoObserve(choiceObs'(a), a, rest, b, h, \pi', v', pr) \wedge \\
&\quad \pi = a; \pi' \wedge v = r(b) + v'.
\end{aligned}$$

In a sense, *BestDoObserve* has the role of *BestDoAux* of DTGolog, except *BestDoObserve* processes possible observations, not nondeterministic actions. In PODTGolog, nondeterministic actions are processed in the *belief update function*:

11. **Observations possible (last)**

After a certain action a and a certain observation o_k, the next belief state is reached. (Here we look at the processing of the last observation associated with the applicable action.) At the time when *BestDoObserve* is called, a specific action, the set of nature's choices for that action and a specific observation associated with the action are under consideration. These elements are sufficient and necessary to update the agent's current beliefs. Inside *BestDoObserve*, the belief state (given a certain action and observation history) is updated via a belief state transition function, similar in vein to the state estimation function (Equation 2.5 of Section 2.1, and also similar to the successor-state axiom for p, the likelihood weight of Bacchus, Halpern and Levesque [2] (cf. Chapter 3)). PODTGolog's belief update function is now defined.

Function *BU(o,a,b)*

1 **foreach** $(s, p) \in b$ **do**
2 $\quad \exists n, s^+, p^+.(s^+, p^+) \in b_{temp} : s^+ = do(n, s) \wedge$
3 $\quad ChoiceNat(n, a, s) \wedge PossAct(n, s) \wedge$
4 $\quad p^+ = p \cdot probObs(o, a, s^+) \cdot probNat(n, a, s) \wedge p^+ \neq 0$
5 $b_{new} = normalize(b_{temp})$
6 **return** b_{new}

A major difference between the POMDP model as defined in Section 2.1 and the POMDP model we define here for the situation calculus, is that here the belief state is not a probability distribution over a *fixed* set of states. If a situation was part of the belief state to be updated, it is removed from the new belief state, and situations that are 'accessible' from the removed situation via $ChoiceNat(n, a, s)$ *and* are executable via $PossAct(n, s)$ are added to the new belief state. Observe that if a potential new situation s^+ has zero

probability, it will not be represented in b_{new}: all situations accessible from s^+ would have zero probability and thus be of no use. To prevent generating, maintaining and reasoning with such impossible situations, we exclude them from belief states.

Because non-executable actions result in situations being discarded, the 'probability' distribution over all the situations in the new belief state may not sum to 1; the distribution thus needs to be normalized:

Function *normalize*(b_{temp})

1 **foreach** unique situation s_t mentioned in b_{temp} **do**
2 $\quad denom = \sum_{(s_t,p_t)\in b_{temp}} p_t \ \wedge$
3 $\quad b_{new} = \{(s_n, p_n) \mid (s_t, p_t) \in b_{temp}, s_n = s_t, p_n = p_t/denom$
4 **return** b_{new}

Note that *SenseCond* (of DTGolog) is not mentioned in the definition of *BestDoObserve* (below). As mentioned in Section 2.2.3, for any action n, $SenseCond(n, \varphi)$ supplies a sentence φ that is placed in the policy being generated. φ holds if and only if φ verifies that action n was the outcome.

PODTGolog takes a different tack: Suppose sv is the value returned by the sensor activated when a is the action performed and that $GetPercept(a, sv)$ is the interface for supplying the value. And suppose $Perceive(a, sv, o)$ verifies whether the sensed value sv can be perceived as the observation o, given a was performed. That is, $Perceive(a, sv, o)$ categorizes the sensor input signal as one of the observations the agent knows about; one can imagine, on encountering $Perceive(a, sv, o)$, the agent says 'Given I performed a and I got percept sv, I think I have perceived observation o'. The need for $Perceive(\cdot)$ is due to sensors possibly returning real valued signals or finely grained discrete signals, but an agent typically does not consider an observation (as an object for reasoning) for each and every gradation of sensor signal.

$ObservationIs(o_k)$ simply *marks* the decision points with the applicable observation. If $GetPercept(\cdot)$ and $Perceive(\cdot)$ supply the test that φ supplies in *BestDoAux*, then we could have defined *BestDoObserve* as (* indicates incorrect definition),

$$
\begin{aligned}
&BestDoObserve^*(\{o_k\}, a, rest, b, h, \pi, v, pr) \doteq \\
&\quad \neg PossObs(o_k, a, b) \wedge \pi = Stop \wedge v = 0 \wedge pr = 0 \vee \\
&\quad PossObs(o_k, a, b) \wedge b' = BU(o_k, a, b) \wedge \\
&\quad \exists \pi', v', pr'.BestDoPO(rest, b', h-1, \pi', v', pr') \wedge \\
&\quad GetPercept(a, sv) \wedge Perceive(a, sv, o_k) \wedge \pi = ObservationIs(o_k)?;\pi' \wedge \\
&\quad v = v' \cdot probNebs(o_k, a, b) \wedge pr = pr' \cdot probNebs(o_k, a, b).
\end{aligned}
$$

However, we opt to perform the tests *outside* the policy. That is, we intend deciding which branch to take at observation choice points outside the policy; the test performed by $GetPercept(\cdot)$ and $Perceive(\cdot)$ is placed outside the policy, in the 'meta-controller'. DT-Golog's *BestDo* supplies policies that are self-contained controllers—no meta-controller is employed. PODTGolog's *BestDoPO* on the other hand, requires a meta-controller to 'interpret' its policies. For our purposes, this is acceptable, because our intent from the start was to embed *BestDoPO* in a larger architecture (meta-controller). Thus, we have the following definition.

$$
\begin{aligned}
&BestDoObserve(\{o_k\}, a, rest, b, h, \pi, v, pr) \doteq \\
&\quad \neg PossObs(o_k, a, b) \wedge \pi = Stop \wedge v = 0 \wedge pr = 0 \vee \\
&\quad PossObs(o_k, a, b) \wedge b' = BU(o_k, a, b) \wedge \\
&\quad \exists \pi', v', pr'.BestDoPO(rest, b', h-1, \pi', v', pr') \wedge \\
&\quad \pi = ObservationIs(o_k)?; \pi' \wedge \\
&\quad v = v' \cdot probNebs(o_k, a, b) \wedge pr = pr' \cdot probNebs(o_k, a, b).
\end{aligned}
$$

With this definition, for each possible observation $o_i \in choiceObs'(a)$ (given an action), $Perceive(a, sv, o_i)$ should be defined by the designer such that it evaluates to true for no more than one of the observations o_i. The policy thus becomes a conditional plan, conditioned on observations. Rens, Ferrein and Van der Poel [85] define this clause of *BestDoPO* with *SenseCond*.

BestDoPO is recursively called with the remaining program and with the horizon h decremented by 1. Also note that the recursive *BestDoPO* will now operate with the updated belief b'. In the definition, $\{o_k\}$ is a single (remaining) observation in the set returned by $choiceObs'$.

12. **Observations possible ($\geq$ 2)**

The following clause defines how the planner processes two or more (remaining) observations in the set returned by $choiceObs'$. The first branch of possible observations is processed, and the other branches in the remainder of the set are processed recursively.

$$
\begin{aligned}
&BestDoObserve(\{o_1, \ldots, o_k\}, a, rest, b, h, \pi, v, pr) \doteq \\
&\quad \neg PossObs(o_1, a, b) \wedge \\
&\quad BestDoObserve(\{o_2, \ldots, o_k\}, a, rest, b, h, \pi, v, pr) \vee \\
&\quad PossObs(o_1, a, b) \wedge b_1 = BU(o_1, a, b) \wedge \\
&\quad \exists \pi', v', pr'.BestDoObserve(\{o_2, \ldots, o_k\}, a, rest, b, h, \pi', v', pr') \wedge \\
&\quad \exists \pi_1, v_1, pr_1.BestDoPO(rest, b_1, h-1, \pi_1, v_1, pr_1) \wedge \\
&\quad \pi = \textbf{if } ObservationIs(o_k) \textbf{ then } \pi_1 \textbf{ else } \pi' \textbf{ endif} \wedge \\
&\quad v = v' + v_1 \cdot probNebs(o_1, a, b) \wedge pr = pr' \cdot probNebs(o_1, a, b).
\end{aligned}
$$

For a full definition of *BestDoPO* as an implementation in Prolog, please consult Appendix B.

4.2 A Simple Example

4.2.1 Example Domain Specification

The example is based on a simple four-state world used in an example in Kaelbling, Littman and Cassandra [48] (see Figure 4.2).

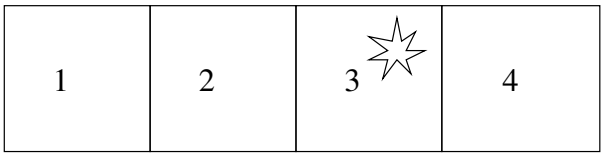

Figure 4.2: Four-state world; four states in a row. Initially the agent believes it is in each state with probabilities [0.04 | 0.95 | 0.00 | 0.01] corresponding to state position.

The star marks the goal location. The formal description follows.

Fluents

- *At*(*lo*, *s*) is true when the agent is at location *lo* in situation *s*, false otherwise;
- *ChoiceNat*(*n*, *a*, *s*) captures whether *n* is nature's choice for action *a* in situation *s*;
- *ChoiceObs*(*o*, *a*, *s*) captures whether *o* is a possible observation given *a* in *s*;
- *probNat*(*n*, *a*, *s*) denotes the probability with which *a* will be realized as *n* in situation *s*;
- *probObs*(*o*, *a*, *s*) denotes the probability with which *a* will result in observation *o* in situation *s*.

Although $ChoiceNat(n, a, s)$, $ChoiceObs(o, a, s)$, $probNat(n, a, s)$ and $probObs(o, a, s)$ are technically *fluents*, they will be treated as rigid predicates, because, in this example, they are situation independent.

States

The four locations are the states that the agent could be in:
$S = \{loc(1), loc(2), loc(3), loc(4)\}$.

Actions

The agent could go left or go right or it could sense its location:
$A = \{left, right, sensloc\}$.

Observations

When the agent tries to sense its location, it could observe one of the four locations; it can also observe nothing (*obsnil*) when executing *left* or *right*:
$\Omega = \{obsloc(1), obsloc(2), obsloc(3), obsloc(4), obsnil\}$.

Precondition axioms

$PossAct(a, s) \equiv a = left \vee a = right \vee a = sensloc$.
That is, it is always possible to go left or right or to sense any location.
$PossObs(obsnil, a, s) \equiv a = left \vee a = right$.
$PossObs(obsloc(z), a, s) \equiv a = sensloc \wedge (z = 1 \vee z = 2 \vee z = 3 \vee z = 4)$.
That is, $\forall a, s.PossAct(a, s) \equiv TRUE$ and $\forall a, s.PossObs(a, s) \equiv TRUE$.

Nondeterminism

- Nature's choice for realizing the agent's intention to move left, is to move the agent left or to move the agent right:
  $ChoiceNat(n, left, s) \equiv n = left \vee n = right$.
- Similarly for when the agent intends to go right:
  $ChoiceNat(n, right, s) \equiv n = left \vee n = right$.
- If the agent intends to sense its location, nature will always allow it to do so:
  $ChoiceNat(sensloc, sensloc, s) \equiv TRUE$.
- When the agent intends to go left or to go right, the agent will not be able to observe anything; nature chooses the null observation for these actions:
  $ChoiceObs(obsnil, a, s) \equiv a = left \vee a = right$.
- When the agent wants to sense its location, it may observe being in any one of the four

locations:
$ChoiceObs(o, sensloc, s) \equiv o = obsloc(x) \wedge (x = 1 \vee x = 2 \vee x = 3 \vee x = 4)$.

State transition function

These fluents capture probabilities as explained earlier. The probabilities are situation independent in this example:

- $probNat(left, left, s) = 0.9$;
- $probNat(right, left, s) = 0.1$;
- $probNat(left, right, s) = 0.1$;
- $probNat(right, right, s) = 0.9$;
- $probNat(sensloc, sensloc, s) = 1$.

Observation function

- $probObs(o, a, s) = p \equiv (a = left \vee a = right) \wedge o = obsnil \wedge p = 1$;
- $probObs(o, a, s) = p \equiv a = sensloc \wedge o = obsloc(x) \wedge [(At(loc(x), s) \wedge p = 0.7) \vee (\neg At(loc(x), s) \wedge p = 0.1)]$.

There are always three locations that the agent cannot be in (for which $\neg At(loc(x), s)$ holds) and one location that it can be in (for which $At(loc(x), s)$ holds). The probability that the agent will perceive that it is in a location that it is actually not in is 0.1; the probability that the agent's perception is correct is 0.7. Therefore, $(3 \times 0.1) + 0.7 = 1$.

Successor-state axioms

$$
\begin{aligned}
&At(loc(x), do(a, s)) \equiv \\
&\quad a = left \wedge (At(loc(x + 1), s) \wedge x \neq 1) \vee (At(loc(x), s) \wedge x = 1) \vee \\
&\quad a = right \wedge (At(loc(x - 1), s) \wedge x \neq 4) \vee (At(loc(x), s) \wedge x = 4) \vee \\
&\quad At(loc(x), s) \wedge (a \neq left \wedge a \neq right).
\end{aligned}
$$

When the robot moves left (right), the robot is in new location x if it was in $x + 1$ ($x - 1$). If the robot is in location 1 and moves left, it will remain in location 1. Similarly if it is in location 4 and wants to move right. Whenever the robot executes an action that is not *left* or *right* (that is, it executes *sensloc*), it remains in the same location.

Reward function

$goal \doteq loc(3)$.
$reward(s) = 5$ **if** $At(goal, s)$, **else** 1; hence, the agent's goal should be location 3.

Sensing conditions

$$
\begin{aligned}
&Perceive(a, sv, o) \equiv \\
&\quad (a = left \vee a = right) \wedge o = obsnil \vee \\
&\quad a = sensloc \wedge \exists x.o = obsloc(x) \wedge \\
&\quad x = 1 \text{ if } sv \text{ is within } 0.123 \text{ of } 3, \text{ else} \\
&\quad x = 2 \text{ if } sv \text{ is within } 0.123 \text{ of } 6, \text{ else} \\
&\quad x = 3 \text{ if } sv \text{ is within } 0.123 \text{ of } 9, \text{ else} \\
&\quad x = 4 \text{ if } sv \text{ is within } 0.123 \text{ of } 12.
\end{aligned}
$$

Imagine a sensor that outputs a measurement value (for example, milli-volts) of 3, 6, 9 or 12 when the robot is respectively in location 1, 2, 3 or 4. The sensor is known to have an error range of 0.123 measurement units above and below the correct measurement. Note that when $a = left$ or $a = right$, it is definitely the case that $o = obsnil$; sv (that is, the sensor) is not consulted for these two actions.

Initial data base

$$
\begin{aligned}
&D_{S_0} = \{At_0, b_0\}. \\
&At_0 = \{At(loc(1), S_1), At(loc(2), S_2), At(loc(3), S_3), At(loc(4), S_4)\}. \\
&b_0 = \{(S_1, 0.04), (S_2, 0.95), (S_3, 0.0), (S_4, 0.01)\}.
\end{aligned}
$$

At_0 denotes that initially the agent can be 'at' any of the four possible locations. Each possibility is a situation: S_1, S_2, S_3 and S_4. b_0 denotes the initial belief state; the probability of each initial situation.

4.2.2 Example Policy Calculation

A simple example follows to illustrate how *BestDoPO* calculates an optimal policy. The same four-state world as depicted in Figure 4.2 is used. All definitions are as above.

Assume, the agent is equipped with the following program; an initial input for *BestDoPO*:

$$BestDoPO(\textbf{while } true \textbf{ do } [left \mid right \mid sensloc],$$
$$\{(S_1, 0.04), (S_2, 0.95), (S_3, 0.0), (S_4, 0.01)\}, 1, \pi, v, pr);$$

the algorithm must compute a one-step optimal policy, that is, for a horizon of 1.

After the iterative component of the program is processed, the following call is made, as per the definition of *BestDoPO* for the nondeterministic choice of actions:

$$\begin{aligned}
&BestDoPO([left \mid right \mid sensloc]; rest, b_0, 1, \pi, v, pr) \doteq \\
&\quad \exists \pi_1, v_1, pr_1.BestDoPO(left; rest, b_0, 1, \pi_1, v_1, pr_1) \wedge \\
&\quad \exists \pi_2, v_2, pr_2.BestDoPO([right \mid sensloc]; rest, b_0, 1, \pi_2, v_2, pr_2) \wedge \\
&\quad ((v_1, left) \geq (v_2, [right \mid sensloc]) \wedge \pi = \pi_1 \wedge v = v_1 \wedge pr = pr_1) \vee \\
&\quad (v_1, left) < (v_2, [right \mid sensloc]) \wedge \pi = \pi_2 \wedge v = v_2 \wedge pr = pr_2)),
\end{aligned}$$

and $BestDoPO([right \mid sensloc]; rest, b_0, 1, \pi_2, v_2, pr_2)$ above, becomes

$$\begin{aligned}
&\exists \pi_3, v_3, pr_3.BestDoPO(right; rest, b_0, 1, \pi_3, v_3, pr_3) \wedge \\
&\exists \pi_4, v_3 4 pr_4.BestDoPO(sensloc; rest, b_0, 1, \pi_4, v_4, pr_4) \wedge \\
&((v_3, right) \geq (v_3, sensloc) \wedge \pi_2 = \pi_3 \wedge v_2 = v_3 \wedge pr_2 = pr_3) \vee \\
&(v_4, right) < (v_4, sensloc) \wedge \pi_2 = \pi_4 \wedge v_2 = v_4 \wedge pr_2 = pr_4)),
\end{aligned}$$

where *rest* is **while** (*true* **do** [*left* | *right* | *sensloc*]). Then the recursive *BestDoPO*s make use of the "Probabilistic observation" definition of the formula. Because–by the action precondition axioms for this example–*left*, *right* and *sensloc* are always executable, the following portion (times three) of the formula is applicable:

$$\exists \pi', v'.BestDoObserve(choiceObs'(left), left, rest, b_0, 1, \pi', v', pr) \wedge \tag{4.1}$$
$$\pi = left; \pi' \wedge v = r(b_0) + v' \tag{4.2}$$

and

$$\exists \pi', v'.BestDoObserve(choiceObs'(right), right, rest, b_0, 1, \pi', v', pr) \wedge \tag{4.3}$$
$$\pi = right; \pi' \wedge v = r(b_0) + v'. \tag{4.4}$$

and

$$\exists \pi', v'.BestDoObserve(choiceObs'(sensloc), sensloc, rest, b_0, 1, \pi', v', pr) \wedge \tag{4.5}$$

$$\pi = sensloc; \pi' \wedge v = r(b_0) + v'. \tag{4.6}$$

From the above, one can see that the action resulting in the largest value v of lines (4.2), (4.4) or (4.6), respectively, will be the 'best' action; the first action recommended by the policy.

For line (4.1) (processing action *left*) the following portion of the "Observations possible" definition is applicable:

$$\begin{aligned}
&b' = BU(obsnil, left, b_0) \wedge \\
&\exists \pi', v'', pr'.BestDoPO(rest, b', 1 - 1, \pi', v'', pr') \wedge \\
&\pi' = ObservationIs(obsnil)?; \pi'' \wedge \\
&v' = v'' \cdot probNebs(obsnil, left, b_0) \wedge \\
&pr = pr' \cdot probNebs(obsnil, left, b_0).
\end{aligned}$$

In this formula (portion), because the recursive call to *BestDoPO* has a zero horizon, $\pi'' = Stop$, and thus $\pi' = ObservationIs(obsnil)?; Stop$. Also, $v'' = r(b')$ because of the definition of *BestDoPO* when $h = 0$. Therefore, the expected value of *left* is $v = r(b_0) + v' = r(b_0) + v'' \cdot probNebs(obsnil, left, b_0) = r(b_0) + r(b') \cdot probNebs(obsnil, left, b_0)$.

The updated belief $b' = BU(obsnil, left, b_0)$ is an input to $v'' = r(b')$. We work out only the first new element of b' in detail (see the definition of $BU(\cdot)$ in Section 4.1.2):

$$(s^+, p^+) \in b_{temp} : s^+ = do(left, s_1) \wedge \ p^+ = 0.04 \times 1 \times 0.9.$$

Because all actions are possible, the only effect that normalization (in the update function) has, is to remove $(do(left, s3), 0.0)$ and $(do(right, s3), 0.0)$ from the new belief state, because of their zero probabilities. $BU(obsnil, left, b_0)$ results in

$$\begin{aligned}
b' = \{&(do(left, S_1), 0.036), (do(right, S_1), 0.004), \\
&(do(left, S_2), 0.855), (do(right, S_2), 0.095), \\
&(do(left, S_4), 0.009), (do(right, S_4), 0.001)\}.
\end{aligned}$$

$r(b_0) = (1 \cdot 0.04) + (1 \cdot 0.95) + (5 \cdot 0.0) + (1 \cdot 0.01) = 1$,
$r(b') = (1 \cdot 0.036) + (1 \cdot 0.004) + (1 \cdot 0.885) + (5 \cdot 0.095) + (5 \cdot 0.009) + (1 \cdot 0.001) = 0.561$ and
$probNebs(obsnil, left, b_0) = (0.04 \cdot 0.9 \cdot 1) + (0.04 \cdot 0.1 \cdot 1) + (0.95 \cdot 0.9 \cdot 1) + (0.95 \cdot 0.1 \cdot 1) + (0.01 \cdot 0.9 \cdot 1) + (0.01 \cdot 0.1 \cdot 1) = 1$.

Therefore, $v = 1 + 0.561 \cdot 1 = 1.561$ and $pr = 1.0$. (Note that $probNebs(obsnil, left, b_0) = 1$—the probability of reaching b'—is expected given action *left*, because there is only one possible successor belief state from b_0 in this case, due to only one possible observation: *obsnil*.)

Now we can instantiate the policy in line (4.2) as:
$\pi = left; (ObservationIs(obsnil)?; Stop) \quad \wedge \quad v = 1 + (0.561)$. Similarly, we can instantiate the policy in line (4.4) as:
$\pi = right; (ObservationIs(obsnil)?; Stop) \wedge v = 1 + (4.424)$.
The expected value of going left is thus 1.561 and of going right is 5.424. More rewards can be expected by the agent if it goes right than if it goes left: $(1.561, left) < (5.424, right)$. But the expected value for choosing *sensloc* is not yet known; we shall now see how PODTGolog finds this value:

For line (4.5) the following portion of the "Observations possible" definition is applicable:

$$b_1 = BU(obsloc(1), sensloc, b_0) \wedge$$
$$\exists \pi'', v'', pr'.BestDoObserve(\{obsloc(2), obsloc(3), obsloc(4)\}, sensloc, rest, b_0, 1, \pi'', v'', pr') \wedge$$
$$\exists \pi_1, v_1, pr_1.BestDoPO(rest, b_1, 1 - 1, \pi_1, v_1, pr_1) \wedge$$
$$\pi' = \textbf{if } ObservationIs(obsloc(1)) \textbf{ then } \pi_1 \textbf{ else } \pi'' \textbf{ endif} \wedge$$
$$v' = v'' + v_1 \cdot probNebs(obsloc(1), sensloc, b_0) \wedge pr = pr' + pr_1 \cdot probNebs(obsloc(1), sensloc, b_0).$$

In this formula (portion), because the recursive call to *BestDoPO* has a zero horizon, $\pi_1 = Stop$, and thus $\pi' =$ **if** $ObservationIs(obsloc(1))$ **then** $Stop$ **else** π'' **endif** . Also, $v_1 = r(b_1)$ because of the definition of *BestDoPO* when $h = 0$. Therefore, the expected value of *sensloc* is $v = r(b_0) + v' = r(b_0) + v'' + r(b_1) \cdot probNebs(obsloc(1), sensloc, b_0)$.

Let $b_2 = BU(obsloc(2), sensloc, b_0), b_3 = BU(obsloc(3), sensloc, b_0)$ and $b_4 = BU(obsloc(4), sensloc, b_0)$. Then it is not too hard to see that $v'' = v''' + v_2 \cdot probNebs(obsloc(2), sensloc, b_0)$, $v''' = v'''' + v_3 \cdot probNebs(obsloc(3), sensloc, b_0)$ and $v'''' = v_4 \cdot probNebs(obsloc(4), sensloc, b_0)$.

But $v_2 = r(b_2), v_3 = r(b_3), v_4 = r(b_4)$. Therefore,

$$\begin{aligned} v &= r(b_0) + v' \\ &= r(b_0) + v'' + r(b_1) \cdot probNebs(obsloc(1), sensloc, b_0) \\ &= r(b_0) + v_4 \cdot probNebs(obsloc(4), sensloc, b_0) + v_3 \cdot probNebs(obsloc(3), sensloc, b_0) \\ &\quad + v_2 \cdot probNebs(obsloc(2), sensloc, b_0) + r(b_1) \cdot probNebs(obsloc(1), sensloc, b_0) \\ &= r(b_0) + r(b_4) \cdot probNebs(obsloc(4), sensloc, b_0) + r(b_3) \cdot probNebs(obsloc(3), sensloc, b_0) \\ &\quad + r(b_2) \cdot probNebs(obsloc(2), sensloc, b_0) + r(b_1) \cdot probNebs(obsloc(1), sensloc, b_0). \end{aligned}$$

Only the calculation of b_1 is shown: $b_1 = BU(obsloc(1), sensloc, b_0) =$

$\{(do(sensloc, s1), 0.028), (do(sensloc, s2), 0.095), (do(sensloc, s3), 0.0),$
$(do(sensloc, s4), 0.01)\}$. After normalizing, this becomes

$$b_1 = \{(do(sensloc, s1), 0.226), (do(sensloc, s2), 0.766), (do(sensloc, s4), 0.008)\}.$$

After calculating $r(b_i)$ and $probNebs(obsloc(i), sensloc, b_0)$ for $i = 1, 2, 3, 4$, we get $v = r(b_0) + \sum_{i=1}^{4} r(b_i) \cdot probNebs(obsloc(i), sensloc, b_0) = 1 + (1 \cdot 0.124) + (1 \cdot 0.67) + (1 \cdot 0.1) + (1 \cdot 0.106) = 1$. Note that $(5.424, right) \geq (1, sensloc)$ meaning that *right* is the action the agent should perform, according to *BestDoPO*.

Finally, the following sentence holds:

$$\begin{aligned}
&BestDoPO(\textbf{while } true \textbf{ do } [left \mid right \mid sensloc],\\
&\quad \{(s1, 0.04), (s2, 0.95), (s3, 0.0), (s4, 0.01)\},\\
&\quad 1,\\
&\quad right; ObservationIs(obsnil)?; Stop,\\
&\quad 5.424,\\
&\quad 1),
\end{aligned}$$

with $right; ObservationIs(obsnil)?; Stop$ being the policy π.

Considering that the agent believed to a relatively high degree that it was initially just left of the 'high-reward' location (3) and its actions are not extremely erroneous, we would expect the agent's first move to be rightwards. Indeed, the policy recommends a rightwards move. Keep in mind that there may be circumstances in which it is more rewarding to sense (for example, to execute *sensloc*) than to move (for example, *left* or *right*).

4.3 Summary

This chapter presented the PODTGolog programming language. It is an extension of DTGolog but for models of the environment that take partial observability into account. An example showed how a domain needs to be specified for PODTGolog, and how *BestDoPO* determines the first action according to the domain specification was shown.

The major disadvantage of using a POMDP planner is its computational complexity. What is gained by dealing with the real world 'head on' by modeling stochastic actions *and* noisy observations, may be lost by the time it takes for a robot to calculate its next move. Some suggestions for optimizing *BestDoPO* are made in Chapter 6. There is a body of work concerned with optimization of POMDPs, however, this is not the focus of the present research.

A robot programming language like ReadyLog, that employs DTGolog's *BestDo* MDP planner, has been implemented in two robots and one agent simulation environment with relative success [28]. It would be interesting to replace *BestDo* with *BestDoPO* on those platforms and compare their performances to the performances with the ReadyLog implementations. Such field experiments must be left for future research.

In the next chapter we shall see how a BDI architecture can take *BestDoPO* to generate policies for stochastic domains. We shall also see how the generic BDI model is instantiated specifically to accommodate *BestDoPO*; vice versa, *BestDoPO* will be slightly modified to be usable by a BDI architecture.

Chapter 5

The Hybrid BDI/POMDP Architecture

In this chapter, an approach is described for combining BDI theory with a POMDP planner (cf. Rens, Ferrein and Van der Poel [86]). Combining the two formalisms can be viewed from two perspectives. One, to enhance an existing planner for use in real-time dynamic domains by incorporating the planner into a BDI agent architecture so that the management of goal selection, planning and replanning is handled in a principled way. Two, to enhance the classical BDI agent architecture by incorporating a POMDP planner into the BDI architecture so that the agent can reason (plan) with knowledge about the uncertainty of the results of its actions, and about the uncertainty of the accuracy of its perceptions. Rens, Ferrein and Van der Poel employ the POMDP planner described in their previous work [85]. In this dissertation, the planner employed is the one defined in the previous chapter, which is almost identical to the one Rens, Ferrein and Van der Poel [86] used. The papers [85, 86] are attached as appendices C and D respectively.

The rest of the chapter has three sections. Section 5.1 shows the basic BDI architecture as previously also presented in [86]. It describes the basic components and processes of the proposed hybrid architecture. Section 5.2 extends the basic architecture to include a notion of meta-reasoning, that is, a notion of controlling the amount and kind of reasoning. Generally, in BDI theory, this notion falls under the term *reconsideration*. The last section discusses some issues concerning the architecture described in this chapter.

5.1 A Basic BDI Architecture employing a Generative Planner

In this section we see how an agent controller in the BDI model can incorporate the *BestDoPO* POMDP planner into its practical reasoning processes. The prototypical control loop of the BDI model was taken as a reference, then it was modified to accommodate planning with POMDP

policies. The proposed architecture is called BDI-POP (BDI with POmdp Planner).

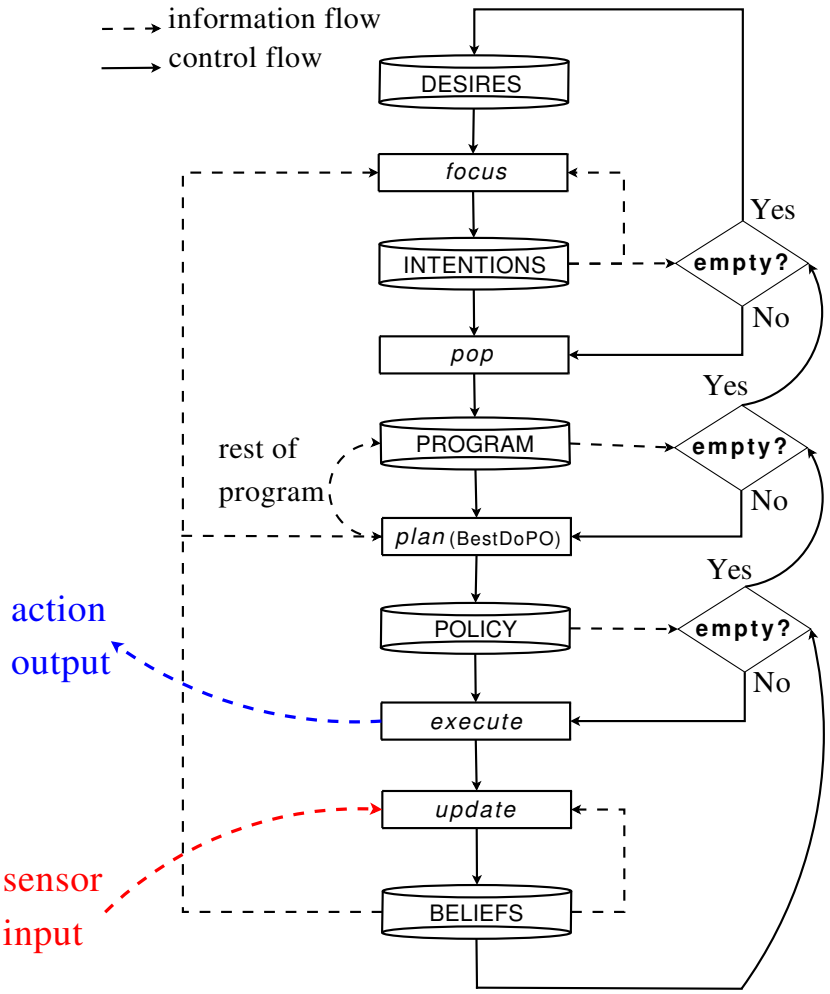

Figure 5.1: Schematic diagram of a sketch of the BDI architecture with the POMDP planner.

First, with the aid of Figure 5.1, some terms and their relationships are introduced. Implicitly included in the "BELIEFS" data store, is the agent's current belief state B, a fixed set of behaviors $\mathcal{B}$ and a fixed set of reward functions $\mathcal{R}$ ($\mathcal{R}$ is considered globally accessible). The "DESIRES" data store contains the fixed set of desires D; the agent's primitive goals; its innate drive. The idea is that each desire refers to a unique goal that the agent is designed to achieve. The *wish* function is omitted from our architecture because the options the agent would pursue at any time are fixed—as D.

Each behavior $b \in \mathcal{B}$ is a quadruple (nom, δ, ach, v): nom is a reference to the Golog program δ, $ach \in D$ refers to the desire that δ can potentially achieve and v is the *value* of δ. The reward functions $r \in \mathcal{R}$ take as argument a reference nom that refers to the program that r is associated with. The following holds: $(\forall d).[d \in D \rightarrow (\exists b).b = (nom, \delta, ach, v) \wedge ach = d]$: for each desire, there exists at least one program to achieve it. Each desire is *characterized* by the set of programs and reward functions that can achieve the desire.

To understand the controller, we also need to consider the agent's deliberation process. *Deliberate*$(\cdot)$ is the procedure that calls and controls the *Focus* predicate and that operates on an *intention stack*: We write $Focus(B, D, I', \mathcal{B}, h^-)$ to be the predicate that selects one $b \in \mathcal{B}$ for each $d \in D$, placing these behaviors in a stack, in ascending order, ordered by the behaviors' *values*. The behavior selected for a desire is the one that can achieve the desire ($ach = d$) and that has the highest value (cf. Algorithm 8). A behavior's value is estimated as the value v of the policy found, generated to a depth h^- : *BestDoPO* is called with $\mathcal{B}$, h^- and the applicable program δ as arguments; v is used and the policy is discarded. We keep $h^- < h$ to save on time spent deliberating. In the definition just below, $h^- = h - 1$, but in general, h^- is constrained by $0 < h^- < h$. *Focus*$(\cdot)$ 'returns' the stack I' of selected behaviors. In $Deliberate(B, D, I, \mathcal{B}, h, i_a, I'')$, i_a is the *active intention*; the intention/behavior popped from the top of the stack I, leaving I'':

$$
\begin{aligned}
&Deliberate(B, D, I, \mathcal{B}, h, i_a, I'') \stackrel{def}{=} \\
&\quad (\exists h^-).h^- = h - 1 \ \wedge \\
&\quad (IsEmpty(I) \wedge (\exists I').Focus(B, D, I', \mathcal{B}, h^-) \ \vee \\
&\quad \neg IsEmpty(I) \wedge (\exists I').I' = I) \ \wedge \\
&\quad (\exists i_a, I'').PopIntentionStack(I', i_a, I'').
\end{aligned}
$$

In terms of Section 2.3.3, when the intention stack is empty, *goal-deep* deliberation occurs, else *intention-deep* deliberation occurs.

Algorithm 8: *Focus*

Input: $B, D, \mathcal{B}, h^-$
Output: I': stack of behaviors; initially empty

```
1 foreach d ∈ D do
2     v_best ← −∞ ;
3     foreach (nom, δ, ach, v) ∈ B s.t. ach = d do
4         BestDoPO(δ, nom, B, h⁻, π, v, pr) ;
5         if v > v_best then
6             v_best ← v ;
7             b_best ← b ;
8     (b_best, v_best) ∈ temp ;
9 Stack(temp, I') ;
```

Algorithm 8 is a top-level pseudocode definition of $Focus(\cdot)$. $Stack(temp, I')$ uses *temp* to order the behaviors in descending values, from top to bottom.

Note that in Algorithm 8, *nom* is a new argument to *BestDoPO*. It is required because each program has its own reward function—*BestDoPO* selects the correct reward function for the current input δ, using *nom* as reference to δ to select the applicable reward function.

BDI-POP tests whether a usable policy could be generated, that is, whether the planner returns

the *Stop* policy: When every outcome of an intended action (according to the input program) is illegal (according to the background action theory), *BestDoPO* returns *Stop*, and we say that the input program is *impossible*. An intention $(nom, \delta, ach, v) \in I$ with δ being impossible is thus defined as an *impossible intention*.

The strategy used in $Deliberate(\cdot)$ to deal with an impossible intention is extremely simple: it is dropped and the next intention on the stack is popped. This is a reasonable strategy because the next intention on the stack has the highest value, and should thus be pursued next. Other strategies are possible, for example, replacing the impossible intention with another intention that achieves the same behavior, if one exists. Calling $Focus(\cdot)$ to refill the intention stack at this time would defeat the principle of *commitment* to intentions.

A logical high-level specification of BDI-POP follows—in three clauses.

1. **Process the program of the active intention**

   If the active intention i_a is (nom, δ, ach, v), then the 'program' of the active intention is δ.

$$
\begin{aligned}
&Agent(B, D, I, \mathcal{B}, h, i_a, \pi) \stackrel{def}{=} \\
&\quad (\exists nom, \delta, ach, v).(nom, \delta, ach, v) = i_a \wedge \\
&\quad (\exists \pi).\pi \neq Stop \wedge \delta \neq nil \wedge \\
&\quad (\exists \pi', a).\pi = a; \pi' \wedge Execute(a) \wedge \\
&\quad (\exists sv).GetPercept(a, sv) \wedge (\exists o).Perceive(a, sv, o) \wedge \\
&\quad (\exists \pi'').GetSubPolicy(\pi', o, \pi'') \wedge \\
&\quad (\exists B').B' = BU(o, a, B)) \wedge \\
&\quad Agent(B', D, I, \mathcal{B}, h, i_a, \pi'').
\end{aligned}
$$

   After the action recommended by the policy is executed, $GetPercept(\cdot)$ returns a sensor value, given the action executed / sensor activated. The agent processes the sensor data and decides what it observed—the agent recognizes the sensor reading via the *Perceive* predicate, which 'outputs' an observation. With this observation, the correct subpolicy is extracted from the current policy, and this (possibly empty) subpolicy becomes the new current policy.

   Then the agent's beliefs must be updated according to what it 'knows' about the effects of its actions. The same belief update function used during planning by *BestDoPO* is used to update the agent's beliefs. The current belief state of the agent will be the 'initial' belief state required as argument to *BestDoPO* the next time the planner is called.

2. **Empty program, empty policy**

   The agent follows the intention with the highest value—the behavior popped from the in-

tention stack; the *active intention*. Initially, the intention stack is empty—that is, $Agent(\cdot)$ is called initially with $i_a = (nil, nil, nil, nil)$, hence $Deliberate(\cdot)$ is called and the active intention is instantiated. Initially, $\pi = Stop$.

If there is no 'rest of program', that is, if the program is empty, $Deliberate(\cdot)$ is called.

$$
\begin{aligned}
&Agent(B, D, I, \mathcal{B}, h, i_a, \pi) \stackrel{def}{=} \\
&\quad (\exists nom, \delta, ach, v).(nom, \delta, ach, v) = i_a \wedge \\
&\quad (\exists \pi).\pi = Stop \wedge \delta = nil \wedge \\
&\quad (\exists i'_a, I').Deliberate(B, D, I, \mathcal{B}, h, i'_a, I') \wedge \\
&\quad Agent(B, D, I', \mathcal{B}, h, i'_a, \pi).
\end{aligned}
$$

3. **Empty policy, program not empty**

Whenever the controller needs a new plan to execute, *BestDoPO* is called to generate a policy with horizon h using the program specified by the active intention. The agent executes the policy until the end of the policy is reached, then *BestDoPO* is called again for the rest of the program.

If the program has become impossible, $Deliberate(\cdot)$ is called.

$$
\begin{aligned}
&Agent(B, D, I, \mathcal{B}, h, i_a, \pi) \stackrel{def}{=} \\
&\quad (\exists nom, \delta, ach, v).(nom, \delta, ach, v) = i_a \wedge \\
&\quad (\exists \pi).\pi = Stop \wedge \delta \neq nil \wedge \\
&\quad (\exists \delta').GetRestProg(\delta, h, \delta') \wedge \\
&\quad (\exists \pi', pr).BestDoPO(\delta, nom, B, h, \pi', v, pr) \wedge \\
&\quad (\exists i'_a).i'_a = (nom, \delta', ach, v) \wedge \\
&\quad (\pi' = Stop \wedge \\
&\qquad (\exists i''_a, I').Deliberate(B, D, I, \mathcal{B}, h, i''_a, I') \wedge \\
&\qquad Agent(B, D, I', \mathcal{B}, h, i''_a, \pi') \vee \\
&\quad \pi' \neq Stop \wedge \\
&\qquad (\exists \pi'', a).\pi' = a; \pi'' \wedge Execute(a) \wedge \\
&\qquad (\exists sv).GetPercept(a, sv) \wedge \exists o.Perceive(a, sv, o) \wedge \\
&\qquad (\exists \pi''').GetSubPolicy(\pi'', o, \pi''') \wedge \\
&\qquad (\exists B').B' = BU(o, a, B)) \wedge \\
&\qquad Agent(B', D, I, \mathcal{B}, h, i'_a, \pi''')).
\end{aligned}
$$

$GetRestProg(\delta, h, \delta')$ assumes that all 'while' conditions in δ are 'true' and not contingent. Thus,

Algorithm 9: *GetRestProgAux*

Input: $\delta, h, Depth$
Output: δ': rest of program, after h actions

```
1  if δ = [(while true do A); B] then
2      GetRestProgAux([A; (while true do A); B], h, Depth, δ') ;
3  else if δ = [E1; E2] and E1 ≠ [E11; E12] then
4      if Depth = h then
5          δ' ← E2 ;
6      else
7          GetRestProgAux(E2, h, Depth + 1, δ') ;
8  else if δ = [E1; E2] and E1 = [E11; E12] then
9      GetRestProgAux([E11; [E12; E2]], h, Depth, δ') ;
10 else
11     δ' = nil ;
```

BDI-POP architectures only allow input programs with 'true' 'while' conditions.
$GetRestProg(\delta, h, \delta')$ is defined by

$$GetRestProg(\delta, h, \delta') \stackrel{def}{=} GetRestProgAux(\delta, h, 1, \delta')$$

and $GetRestProgAux(\cdot)$ is defined in Algorithm 9.

Given our present definition of *Deliberate*$(\cdot)$, to guarantee that the agent deliberates at regular intervals, the designer must allow only finite programs for achieving intentions, Hence, fixing the interval period—to the degree that intentions become impossible. Adding a *Reconsider* predicate that tells the agent once every control cycle whether to deliberate is a more sophisticated method. One possible definition of the *Reconsider* predicate is discussed in the next section, and also shown, is how BDI-POP is modified to accommodate *Reconsider*.

It can be seen that BDI-POP employs a *single-minded* commitment (intention maintenance) strategy.

5.2 Adding Reconsideration to the Architecture

Reconsideration was discussed in detail in Section 2.3 and we saw the important role it plays in BDI theory. Therefore, adding reconsideration to the basic architecture developed in the previous section is investigated here. The resulting architecture will be called BDI-POP(R).

In Algorithm 5 in Chapter 2, the *reconsider* function is placed just after the belief *update* function in the agent control loop. It is placed in the same relative position of the BDI-POP

control loop. To be precise, clauses 1 and 3 of the definition of *Agent* presented in the previous section, change as follows (clause 2 remains unchanged):

1.

$$
\begin{aligned}
&Agent(B, D, I, \mathcal{B}, h, i_a, \pi) \stackrel{def}{=} \\
&\quad \ldots \\
&\quad (\exists B').B' = BU(o, a, B)) \wedge \\
&\quad (Reconsider(B', I, i_a, h, F_V, F_I) \wedge \\
&\qquad Deliberate(B', D, I, \mathcal{B}, h, i'_a, I') \wedge Agent(B', D, I', \mathcal{B}, h, i'_a, \pi'') \vee \\
&\quad \neg Reconsider(B', I, i_a, h, F_V, F_I) \wedge \\
&\qquad Agent(B', D, I, \mathcal{B}, h, i_a, \pi'')).
\end{aligned}
$$

3.

$$
\begin{aligned}
&Agent(B, D, I, \mathcal{B}, h, i_a, \pi) \stackrel{def}{=} \\
&\quad \ldots \\
&\quad (\exists B').B' = BU(o, a, B)) \wedge \\
&\quad (Reconsider(B', I, i'_a, h, F_V, F_I) \wedge \\
&\qquad Deliberate(B', D, I, \mathcal{B}, h, i''_a, I') \wedge Agent(B', D, I', \mathcal{B}, h, i''_a, \pi''') \vee \\
&\quad \neg Reconsider(B', I, i'_a, h, F_V, F_I) \wedge \\
&\qquad Agent(B', D, I, \mathcal{B}, h, i'_a, \pi''')).
\end{aligned}
$$

Algorithm 10 is a high-level definition of *Reconsider*$(\cdot)$. Observe that I'_o is the original stack of intentions, without the active intention; then I_o (line 1) is the complete intention stack.

In the algorithm, *IntnStack_CurrentValues*(B', I_o, h^-) (line 4) returns a stack of intentions I'_c such that, for each $(nom, \delta, ach, v_o) \in I_o$, $(nom, \delta, ach, v_c) \in I_c$, such that, *BestDoPO*$(\delta, nom, B', h^-, \pi, v_c, pr)$. Let I_c (line 6) be the intentions (including the active intention) with values as found through *BestDoPO* with the *updated* belief state B'.

Furthermore, *Reconsider*$(\cdot)$ takes the designer supplied arguments F_V and F_I—let $0 < F_V, F_I \leq 1$. F_V is a factor for lowering the value of the average value of I_c so that re-deliberation is considered only if the average value of I_o (with i_a) is less than the average value of I_c times the factor F_V (lines 7–9). This is to prevent the agent from reconsidering, when there are relatively small variations in the average value of its intentions, according to its current beliefs.

Notice that I_o and I_c may have almost equal average values, even though the order of the two stacks may be rather different. An agent should not want to keep I_o as its intentions when

Algorithm 10: Reconsider

Input: $B', I'_o, i_a, h, F_V, F_I$
Output: **true** or **false**

1 $I_o \leftarrow Push(i_a, I'_o)$;
2 $(nom, \delta, ach, v_o) = i_a$;
3 $h^- \leftarrow h - 1$;
4 $I'_c \leftarrow IntnStack\_CurrentValues(B', I_o, h^-)$;
5 $BestDoPO(\delta, nom, B', h^-, \pi, v_c, pr)$;
6 $I_c \leftarrow Push((nom, \delta, ach, v_c), I'_c)$;
7 $A_{orig} \leftarrow AverageValue(I_o)$;
8 $A_{curr} \leftarrow AverageValue(I_c)$;
9 $A'_{curr} \leftarrow A_{curr} \times F_V$;
10 $N \leftarrow NuofInversions(I_c, I_o)$;
11 $M \leftarrow MaxPossInversions(I_o)$;
12 $M' \leftarrow M \times F_I$;
13 **if** $A_{orig} < A'_{curr}$ *or* $N \geq M'$ **then**
14 **return true**
15 **else**
16 **return false**

the order of these intentions is irrational. That is, if the agent had to reorder the intentions in stack I_o according to its current updated beliefs to get I_c, then the agent may want to consider re-deliberating if the order of the intentions has changed 'sufficiently'. Define "sufficiently" as: when the number of inversions of I_c with respect to I_o is greater than/equal to the maximum number of inversions possible of the intentions in I_o, times the designer supplied factor F_I (lines 10–12).

The *Reconsider* function thus considers both a sufficient drop in the average value of intentions (a *value-based* reconsideration strategy) *and* a sufficient amount of difference in the order of the intentions with respect to the current belief state (a *knowledge-based* reconsideration strategy)[1] (at line 13 of Algorithm 10).

5.3 Discussion

A somewhat significant difference of our hybrid architecture from the perspective of control via POMDP policy generation, is that—as stand-alone controller—the POMDP planner takes a single plan with a single associated reward function, to generate a policy. The new hybrid architecture, on the other hand, takes several programs, each with an associated reward function. This aspect of the agent being able to reason over multiple behaviors has the advantage that the agent designer can separately specify behaviors that should—at least intuitively—be considered separately.

[1] Reconsideration strategies were defined in Section 2.3.3.

BestDoPO expands Golog programs into hierarchically structured plans (policies), and only programs that have been selected as intentions are expanded into policies. Each program can generate a policy—or several policies if the program is expanded piece-wise. Viewing a policy tree as an intention structure in the sense of traditional BDI architectures, each program in the intention stack represents at least one intention structure. BDI-POP, therefore, maintains *several* unexpanded intention structures, only expanded when popped from the intention stack.

In BDI-POP, intentions and intention structures are not partial and hierarchical—as they traditionally are in BDI architectures—in the sense that intentions are plans that may be composed of more abstract intentions referring to more concrete plans. The program δ in the intention/behavior (nom, δ, ach, v) mentions only (primitive) actions, not other intentions/behaviors. Allowing for partial/hierarchical intentions in BDI-POP is left for future work.

More sophisticated reconsideration strategies (*reconsider* functions) are possible, but investigation in this direction are beyond the scope of this dissertation.

Chapter 6

Ideas for Efficiency

Optimal POMDP solvers are known to be highly inefficient. Four different ideas are explored here, to make the BDI-POP POMDP-planner more efficient in terms of time and memory usage. The methods are divided into two categories: (i) those that reduce belief state dimensionality, and (ii) those that reduce decision tree branching factor. In the former category we discuss

1. Culling situations (represented in any belief state) with associated probability less than some value;
2. Condensing belief states: transforming (and maintaining) belief states representing situations, to belief states with situations representing a fixed set of world states.

In the latter category we discuss

3. Constraining the number of belief states to a fixed amount at any tier of the decision tree, using the probability of reaching a state;
4. Constraining the number of belief states to a fixed amount at any tier of the decision tree, using state utility.

Either of the two items from category (i) can be combined with either of the items from category (ii), yielding four combinations. In other words, one could apply one of the four individual methods, or one could apply four mixtures of methods.

Results of experiments run on implementations of category (i) methods are presented in Chapter 7.

6.1 Culling Situations

6.1.1 By Probability

The approach taken here is very simple. In the belief update function, remove any situation represented in a belief state that has a probability less than some predefined probability value. The reasoning is that an agent should not further consider, as part of its beliefs, situations that are very unlikely. Then normalize the remaining situations so that they sum to 1. If a predefined probability value is c (c could be set to 0.01 for example), the belief update function becomes

Function *BU(o,a,c,b)*

foreach $(s, p) \in b$ **do**
  $\exists n, s^+, p^+.(s^+, p^+) \in b_{temp-1} : s^+ = do(n, s) \wedge$
  $ChoiceNat(n, a, s) \wedge PossAct(n, s) \wedge$
  $p^+ = p \cdot probObs(o, a, s^+) \cdot probNat(n, a, s) \wedge p^+ \neq 0$
$b_{temp-2} = normalize(b_{temp-1})$
foreach $(s^+, p^+) \in b_{temp-2}$ **do**
  $(s^+, p^+) \in b_{temp-3} : p^+ > c$
$b_{new} = normalize(b_{temp-3})$
return b_{new}.

At first glance, the reader may wonder why the line $p^+ = p \cdot probObs(o, a, s^+) \cdot probNat(n, a, s) \wedge p^+ \neq 0$ is not simply changed to $p^+ = p \cdot probObs(o, a, s^+) \cdot probNat(n, a, s) \wedge p^+ > c$ and then let $b_{new} = normalize(b_{temp1})$ be the last line defining the function. Suppose $b = \{(do(a_1, s), 0.005), (do(a_2, s), 0.995)\}$, and suppose n_1 and n_2 are outcomes of a^+, and $PossAct(n_1, s)$ is true, but $PossAct(n_2, s)$ is false. Then with the naive approach, if $c = 0.01$, then $BU(o, a^+, c, b) = b_{new} = \{\}$, whereas using a two step process—culling and normalizing twice—
$b_{temp-1} = \{(do(a^+, do(a_1, s)), 0.005)\}$, $b_{temp-2} = \{(do(a^+, do(a_1, s)), 1.0)\}$ and
$b_{new} = \{(do(a^+, do(a_1, s)), 1.0)\}$. Using the two-step process thus reduces the chance of an agent becoming 'unconscious' (having an empty belief state), especially when the belief state has only few situations in it.

For later reference, we call this (two-step) method *belief state reduction by probability cut-off*.

6.1.2 By Probability and Dimensionality

In any belief state, consider only the x most likely situations. The idea here is very similar to the idea of the method above; here though, the size (number of situations represented) of a belief state is limited to x. (Belief state dimensionality is unlimited in the previous method.)

Intuitively, it seems reasonable to consider the most likely situations only; this is the thinking of both methods in category (i). The present method considers computation as well, by limiting state size. The advantage of the present method is that no 'cut-off' probability (c) needs to be chosen or determined: Agents in different domains may perform differently for the same value of c. However, the disadvantage of the present method is that the choice of x's value must be justified.

There is another problem to applying the idea naively: Suppose a belief state b mentions 100 situations, 10 situations have probability 0.02, 20 have probability 0.01 and the other 70 situations make up for the remaining probability mass, but all with probabilities less than 0.01. Now if x is 20, the 10 situations with probability 0.02 will be retained. But which of the 20 with probability 0.01 must be retained? In this work, this complication is ignored. For later reference, we call this simple method *belief state reduction by x most likely*. The method is defined below.

Function *BU(o,a,x,b)*

foreach $(s, p) \in b$ **do**
  $\exists n, s^+, p^+.(s^+, p^+) \in b_{temp-1} : s^+ = do(n, s) \wedge$
  $ChoiceNat(n, a, s) \wedge PossAct(n, s) \wedge$
  $p^+ = p \cdot probObs(o, a, s^+) \cdot probNat(n, a, s) \wedge p^+ \neq 0$
if $SmallEnough(b_{temp-1}, x)$ **then**
  $b_{temp-3} = b_{temp-1}$
else
  $b_{temp-2} = sortDescending(b_{temp-1})$
  $b_{temp-3} = popFirst(x, b_{temp-2})$
$b_{new} = normalize(b_{temp-3})$
return b_{new}.

$SmallEnough(b_{temp-1}, x)$ is true if and only if the number of situations in b_{temp-1} is less than/equal to x. $sortDescending(b_{temp-1})$ sorts the situations in b_{temp-1} in order of descending probability. $popFirst(x, b_{temp-2})$ returns the first x situations (with the highest probability).

6.2 Condensing Belief States

Suppose an agent lives in a world with 25 states. The agent's initial belief state may contain any number of situations less than or equal to 25 to describe in which states it thinks it is in initially. For example, it may believe it is in states $1, 2, 3, 4$ and 5, each with probability 0.2. Say all the agent's stochastic actions have three (always executable) outcomes, then its next belief state will contain $5 \times 3 = 15$ situations, and the next belief state $15 \times 3 = 45$ situations, and so on.

Several situations may possibly describe the same state, the states are simply reached via different action histories. Refer to this as the *situation equivalence problem* in situation calculus. If one could combine such similar situations, one could constrain the agent's beliefs to be a probability distribution over a maximum of—in this example—25 situations (states), each such situation having a probability equal to the sum of the probabilities of the situations that were combined. Each new combined situation represents one state and the situation has the whole probability of the agent being in that state. There is then a one-to-one correspondence between situations and states. The new belief update function would thus look like this:

Function *BU(o,a,b)*

foreach $(s, p) \in b$ **do**
  $\exists n, s^+, p^+.(s^+, p^+) \in b_{temp-1} : s^+ = do(n, s) \wedge$
  $ChoiceNat(n, a, s) \wedge PossAct(n, s) \wedge$
  $p^+ = p \cdot probObs(o, a, s^+) \cdot probNat(n, a, s)$
$b_{temp-2} = normalize(b_{temp-1})$
foreach $state \in \Omega$ **do**
  $\exists s_n, p_n.(s_n, p_n) \in b_{new} : NewSituation(s_n) \wedge$
  $\forall s_t, p_t.(s_t, p_t) \in b_{temp-2} \wedge Similar(s_t, state) \wedge \forall F.F(s_n) \equiv F(s_t) \wedge$
  $p_n = \sum_{\forall s_t, p_t.(s_t, p_t) \in b_{temp-2}} p_t$
return b_{new}.

We call this the *belief state condensation by transformation* method of belief update.

The agent designer must be careful to specify actions and states such that states are closed under actions. In the function above, each time the agent's belief state is updated, a new unique situation term s_n is created, such that $NewSituation(s_n)$ holds; one for each state in Ω . F is a fluent. In $Similar(s_t, state)$, s_t is universally quantified, $(\forall s_t)[Similar(s_t, state)]$ iff $(\forall F)[F(s_t)$ & predicate F holds in *state*]. The second order formula $(\forall F)F(s_n) \equiv F(s_t)$ ensures that all the fluents that held in all the similar situations s_t hold in the new situation s_n representing *state*—and the fluents hold *only* then.

With this modification, belief states are *progressed*. The idea is similar to the progression of a KB (cf. [84], Chapter 9), but here—for each belief state—we are progressing several KBs. Also, we *maintain* the progressed belief states, which leads to loss of memory. Memory is not necessary while planning though: the belief states will be discarded once a policy has been generated. Nevertheless, the agent can still retain the memory of the actions it *actually* performed.

6.3 Branch Pruning by Reachability

The branching factor in a decision tree is normally $a \times o$, where a is the number of actions the robot can choose from and o is the average number of observations per action. The search space for generating a policy thus grows exponentially by a factor of $a \cdot o$ in the depth of the policy (the planning horizon h). During an agent's activity in the world, certain belief states may never be reached or entertained because the observations involved in reaching some state may occur with tiny probabilities. It may therefore be wasted effort during planning to consider sub-decision-trees rooted at belief states that occur with such low probability.

We could thus decide to entertain only the m most likely belief states per decision tree tier while searching for a policy. The decision tree size—in the number of belief states—will be approximately $m \times h$ when its generation is complete. The size of such decision trees is thus constant.

To determine the probability of any belief state b, one needs to consider the probabilities of all the actions and observations on the path to b. Suppose we are looking for a policy of depth 3. Consider some leaf belief state b'''; let the path to b''' have the sequence $(a', o', a'', o'', a''', o''')$ of actions and observations on it. Then the *a priory* probability of the robot being in b''' is $Pr(a') \cdot Pr(o') \cdot Pr(a'') \cdot Pr(o'') \cdot Pr(a''') \cdot Pr(o''')$. Let $Pr(o) = probObs(o, a, b)$, where b is the appropriate belief state for the observation in the tier of the decision tree. For example, $Pr(o') = probObs(o', a', b_0)$, where b_0 is the initial belief state. Because one does not know beforehand what action the robot will choose, we assume equal likelihood for all actions being considered at that time. For example, if at some belief state the agent can choose between four actions, then $Pr(a) = 0.25$ for each action.

We only consider actions and observations that are *possible*. Therefore, out of all actions the robot can entertain, say $\{a_1, \ldots, a_n\}$, if $PossAct(a_1, b)$ is false, let the probability of each remaining action being chosen be equal to $1/(n-1)$.

To implement this idea, one needs all belief states in a tier of the decision tree to be available, so that a set m of 'most likely' belief states can be determined—there is one set m for each tier. That is, there is one set m that is updated/maintained during the generation of a decision tree. To accomplish this, change the search method of the planner from depth first to breadth first. Theoretically this should not cause memory problems, because the tree generated, grows linearly; linear growth is the aim of this modification.

Changing from a depth first to a breadth first semantics with the incorporation of the m most likely set of belief states, is not straight forward, and is left for future work.

6.4 Branch Pruning by Utility

A method similar to that in Section 6.3—choosing the 'best' m belief states per tier to expand—is now presented. The only difference is the definition of 'best'; here 'goodness' of a state is proportional to its utility.

Suppose the planning horizon is $h = H$. Hence a decision tree of depth H (H tiers) must be generated for the required number of actions. Determining the belief state utilities (total expected rewards) optimally, would require the *whole* decision tree to be expanded to the depth H. But expanding the whole decision tree is exactly what we are trying to avoid!

One solution would be to expand the decision tree to depth 1 (assuming $h > 1$), then determining the x most valuable belief states. Then expand only these m belief states by one tier, and select the x most valuable belief states again, and so on. This solution provides most linearity/tractability in the generation of a decision tree. But it is least accurate/optimal, because expected values are being estimated—optimal paths in the tree might be pruned due to selection with incomplete information.

An *ad-hoc* approach might be to select depth $h' = \lceil H/3 \rceil$ for expanding belief states. For example, if $h = 9$, then $h' = 3$, and after generating the decision tree for three action tiers, select the x belief states with highest utilities. Expand only these x states for a further three action tiers, and again select the m belief states with highest utilities, till depth 6. Finally, expand these m belief states till depth 9. From the pruned decision tree generated in this fashion, determine the policy.

Suppose there are 10 actions and, on average 5 observations associated with each action, and all actions and observations are always possible. Then a complete belief decision tree of horizon 9 (without pruning) has $1+(10\times5)+((10\times5)\times(10\times5))+(10\times5)^3+50^4+\ldots+50^9 = \sum_{i=0}^{9} 50^i$—$O(50^9)$ belief states. With the pruning method, using $h' = 3$, the number of belief states generated will be $1+(10\times5)+(10\times5)^2+(10\times5)^3+50x+50(50x)+50(50(50x))+50x+50(50x)+50(50(50x)) = 1 + 50 + 50^2 + 50^3 + 2(50 + 50^2 + 50^3)x$—$O(50^3)$. This is an improvement by an exponential factor of 3 in this example.

Again, implementing and testing this method is not pursued now.

6.5 Discussion

The *belief state reduction by probability cut-off* method is very simple to implement, and it will be seen in the next chapter to yield improved results. It is however, to some degree, *ad hoc*. The method of *belief state reduction by x most likely* seems to be no better as far as principled motivation goes. Its implementation is also, relatively speaking, computationally

more intensive, but nevertheless also delivers improved results. The method of *belief state condensation by transformation* has a good formal basis, however, much more computation is required than the other category (i) methods to implement the method. See the next chapter for the efficacy of the method.

Although the 'branch pruning by utility' method has higher computational complexity than the other category (ii) method, pruning by utility instead of reachability may result in more effective policies being generated. A comparison of the effectiveness of the two category (ii) methods, by observing the behavior of agents employing the respective methods, may yield insights into these optimization approaches.

All the methods cause policies to be sub-optimal. Sub-optimal policies imply sub-optimal agent behavior in terms of the value of policies, *per policy*. However, for agents that repeatedly generate and execute policies, if policy generation can be sped up, their reactivity increases and they may perform better in the *long-run*, even with sub-optimal policies.

Be aware that there is potential for explosion of memory requirement for decision trees both in the requirements for representing belief states *and* in the number of belief states that need to be stored. Therefore, to curb exponential growth of the size of decision trees, with respect to h, optimization must be applied to both size of *and* number of belief states.

Chapter 7

Experimentation

This chapter assesses the performance of *BestDoPO* and various architectures employing *BestDoPO* under various conditions. The first section justifies the method of assessment and mentions the assumptions made with respect to the experimental framework. The next section focuses on the performance of the planner alone. Section 7.3 introduces the simulation world that will be used in Sections 7.5 and 7.6. A 'base-line' architecture is described in Section 7.4 to help gauge the performance of BDI-POP and BDI-POP(R) in Section 7.6. The results are discussed in the final section. For a discussion of the results in a broader context, please refer to the concluding chapter. (In the rest of the dissertation, BDI-POP((R)) means BDI-POP *and* BDI-POP(R).)

7.1 Method and Assumptions

There seems to be only three ways in which one can assess the performance of *BestDoPO* and BDI-POP((R)): (1) analytically/mathematically, (2) empirically through simulation, and (3) empirically through real-world implementation. Although an analytical assessment of *BestDoPO* is conceivable, it is beyond the scope of this dissertation. An analytical assessment of BDI-POP((R)) would not be conceivable to most researchers, and definitely not within the scope of the current work.

Although empirical assessment "through real-world implementation" is preferred above empirical assessment "through simulation", much resources are required for the former approach. Experimentation using a simulator was thus the option chosen. It was decided to create a simulator in the Prolog programming language. Moreover, creating the simulator from scratch allowed for the flexibility of creating a world suited to the current work. We acknowledge that creating a simulator from scratch imposes constraints, including great difficulty in simulating detailed physical conditions and of complex, detailed scenarios. Nevertheless, the FireEater-

world simulator was created and implemented in Prolog as described in Section 7.3.

7.2 Optimization Methods with the *BestDoPO* Planner Alone

We look here at experimental results of three optimization methods defined in Chapter 6. Instead of FireEater-world, a simpler experimental environment is first used to notice effects of time on policy generation and on the quality of policies.

In this section we only investigate the *BestDoPO* planner's performance (relative to the agent's action and world models) independent of the agent's performance on some task in an actual (virtual) domain. Actually, there is no *agent* as such under consideration here, because there is no architecture for the planner to be embedded in.

In the experimental environment of this section, there is no random element; only one run is thus necessary per setting of a parameter.

Plans must be generated for an agent inhabiting a 5×5 grid world, where the agent's task is to move towards the cell that contains the (only) star. In this toy world, there are no obstacles or anything to distract an agent. The domain specification is very similar to the specification in Section 4.2.1.

We call

$$BestDoPO(\textbf{while } \neg AtGoal \textbf{ do } [left \mid right \mid up \mid down \mid sensloc], b_0, h, \pi, v, pr),$$

where h will be set according to the experiment design.

The reward function used in this section is essentially designed such that the agent gets rewards proportional to the inverse of the Manhattan distance from the star.

7.2.1 Experiment 1

First, to show that *BestDoPO* can deal with the case where an agent is initially uncertain where it is, a policy of depth 4 (that is, $h = 4$) is sought, using the 'vanilla' belief update function. The initial belief state is $b_0 = \{(s_9, 0.9), (s_{13}, 0.05), (s_{17}, 0.05)\}$ and the following are in the agent's knowledge base: $At(loc(2,4), s_9)$, $At(loc(3,3), s_{13})$ and $At(loc(4,2), s_{17})$. See Figure 7.1. The star is at grid location (1,1) ($loc(1,1)$).

The policy was (essentially) $\ell; d; d; d$—*left* is abbreviated as ℓ and *down* as d. Note that this is a policy one would expect if the agent believes strongly (90%) that it is in cell (2,4).

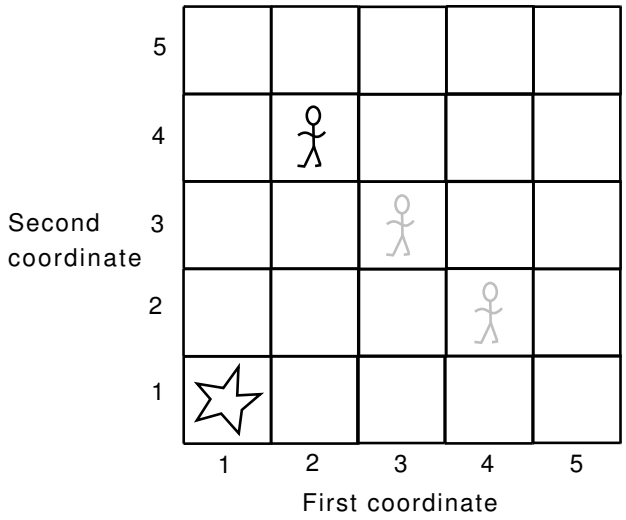

Figure 7.1: A 5×5 grid world. The goal is to reach the star. The agent is initially possibly in three situations.

7.2.2 Experiments 2, 3 and 4

Figure 7.2 indicates that the agent's initial belief state is $b_0 = \{(s_{19}, 1.0)\}$, where $At(loc(4,4), s_{19})$ is in the agent's knowledge base. Because the agent has absolute certainty about its initial state (that is, situation s_{19}), it is simple to classify a policy as "optimal" or "non-optimal" by inspection.

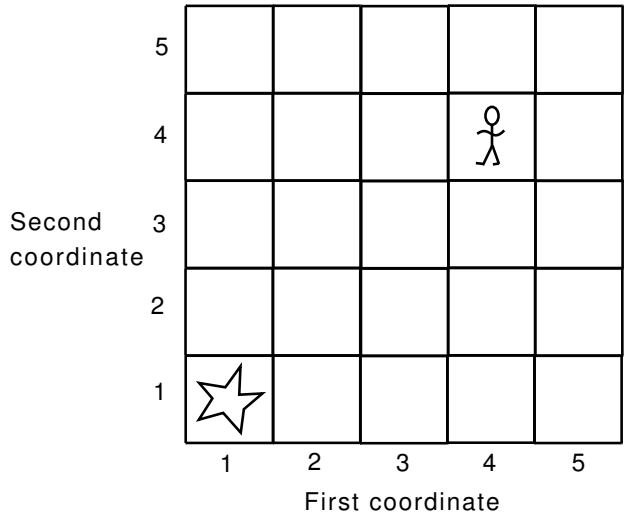

Figure 7.2: A 5×5 grid world. The goal is to reach the star. The agent is certain of its initial situation.

For the following three batches of experiments, a policy π for six actions (that is, $h = 6$) is sought.

Using the 'vanilla' belief update function, the running time was 13346 seconds and the policy was (essentially) $d; d; d; \ell; \ell; \ell$. Note that this is an optimal policy. Now the following two optimization methods can be assessed against this base-line result.

Cut-off probability	Running time (sec.)	Policy found	Optimal?
$c = 0.1$	916	$d; d; d; \ell; \ell; \ell$	Yes
$c = 0.01$	4764	$d; d; d; \ell; \ell; \ell$	Yes
$c = 0.001$	5648	$d; d; d; \ell; \ell; \ell$	Yes

Table 7.1: Results of policy generation using belief state reduction by culling situations below a cut-off probability.

Table 7.1 gives the results of using the *belief state reduction by probability cut-off* method, where the probability cut-off parameter c is varied between 0.001 and 0.1. This seems to be a reasonable range. Culling all situations reachable with a probability less than or equal to 0.5 or 0.00001, for example, seems unreasonable. Showing for what values of c this optimization

method has the most and least effect, is not the aim of this study. We only show that the method has significance.

Situations retained	Running time (sec.)	Policy found	Optimal?
$x = 1$	885	$d; d; d; \ell; \ell; \ell$	Yes
$x = 11$	7704	$d; d; d; \ell; \ell; \ell$	Yes
$x = 21$	10374	$d; d; d; \ell; \ell; \ell$	Yes

Table 7.2: Results of policy generation using belief state reduction by retaining the x most likely situations.

Table 7.2 gives the results of using the *belief state reduction by x most likely* method, where x is varied between 1 and 21.

Similar to the experiment about the previous optimization method, here, only three settings are chosen to best illustrate the potential of this method. $x = 1$ is the smallest possible setting. And setting $x \geq 21$ would not shed much light on the subject: 13346 seconds running time is the worst case and 10374 for $x = 21$ is close to that.

7.2.3 Experiment 5

A fifth batch of experiments compares the 'vanilla' belief update function and the *belief state condensation by transformation* method, for $h = 1, 2, 3, 4, 5$ and 6 (cf. Tbl. 7.3). The *belief state condensation by transformation* method yields an optimal policy when $h = 6$.

Horizon	Running time (sec.) 'vanilla' method	Running time (sec.) modified method
$h = 1$	0	0
$h = 2$	0	0
$h = 3$	1	2
$h = 4$	20	32
$h = 5$	515	502
$h = 6$	13346	7755

Table 7.3: Results of policy generation using belief state condensation by transforming situations into states.

With this method, there is no parameter to set, unlike the previous two methods. In pilot trials, it was found that this method only starts producing significant results for relatively far horizons; in this case, for $h \geq 6$. Table 7.3 makes this clear.

7.2.4 Analysis

Experiment 1 verifies that the basic or 'vanilla' POMDP planner works as expected in a simple problem, given an uncertain initial situation.

Experiments 2,3, and 4 tested the 'vanilla' planner, the planner with *belief state reduction by probability cut-off*, and respectively, the planner with *belief state reduction by x most likely*. For the two experiments where optimization is employed, in the best cases, the improvement over the non-optimized planner is by a factor of fifteen.

The significance of the improvement seen in experiments 3 and 4 should encourage an agent designer to always employ a planner with one of the optimization methods—if programming in PODTGolog.

The result that policies were always optimal, is surprising, especially in the case where the *belief state reduction by x most likely* method was used, and $x = 1$. $x = 1$ means that only one situation is maintained in the agent's belief states. It might be, that an optimal policy is achieved, even under this most strict constraint, because of the simplicity of the simulation world.

In the experiment with *belief state reduction by probability cut-off*, less than half the running time used by the 'vanilla' planner is used, even though only situations with probability less than 0.001 are culled. This result indicates that there is potential that much information can be retained in the agent's belief states, while still improving planning running time—if this optimization method is employed.

Use of the *belief state condensation by transformation* method achieves a halving of running time when $h = 6$. The results in Table table:vanil-vs-cond, however, suggest that significant improvement only occurs from $h \geq 6$ in this experimental case. Thus, if shorter policies are periodically sought—as in Section 7.4—this optimization method may not have the impact that the other two methods may have. Moreover, the other two methods allow for a desired level in trade-off between policy accuracy and planner running time, due to the fact that these methods have parameters that can be set.

More is said in the next chapter about *BestDoPO* in practice. In the rest of this chapter, the performance of the planner *together* with a 'meta-controller' are evaluated.

7.3 The FireEater-world Simulation Environment

To validate the BDI-POP architecture and to gain a sense for its performance potential, we observe the behavior of an agent based on the architecture, in a simulation. The simulation environment is inspired by Tileworld [77], a testbed for agents. FireEater world was designed and

implemented. It is a dynamically changing grid world in which an agent is situated. There are obstacles that change position, fires that can be 'eaten' and 'power packs' that can be collected.

The agent gets one 'point' for eating one fire and three 'points' for collecting a power pack. It can only eat a fire or collect a power pack if it is in the same cell as the fire/power pack. The agent can go left, right, up or down—locomotive actions which are stochastically nondeterministic; it can also sense its location (probabilistically) and do nothing (deterministically). To summarize, the agent has eight actions available to it: $up, down, left, right$, $noop$ (do nothing), *eat* (fire), *grab* (power pack) and *sensloc* (sense location).

Once a fire is consumed or a power pack collected, it does not re-appear, however, their initial locations are random. There is initially always 36 fires and power packs in sum, the ratio of which—the *fire:pack* ratio—may be chosen as a simulation parameter: 9:27, 18:18 or 27:9.

There are always 24 obstacles (throughout the simulation) and their location changes during a simulation run. The rate at which they change is a simulation parameter; the simulator asks for the number of obstacles whose position must change per simulation cycle. Call this parameter the *rate of change* (RC). An agent cannot move into/through a cell that has an obstacle in it.

Similarly, the simulator expects as input a parameter for the number of agent actions allowed per simulation cycle. Call it *agent speed* (AS). *Dynamism* is defined as RC/AS.

Let t be a trial (simulation run) and b a batch of trials. Let p_t be the number of points collected in some trial t. *Effectiveness* is the average points collected per trial for a batch of trials ($\sum_{t \in b} p_t / |b|$). Let r_t be the running time of trial t in seconds. *Efficiency* is the number of points collected per minute for a batch of trials ($(\sum_{t \in b} p_t \times 60) / \sum_{t \in b} r_t$). As the major performance measure, we define $\mathcal{P} = Effectiveness + Efficiency$.

A last parameter for the simulator still needs defining: *simulation time* (ST). When the local counter (LC) of the simulator reflects that more time has passed than represented by ST, the simulator halts. For all trials (for all experiments), $ST = 20$. Note however, that if $LC < ST$ in some cycle, in the next cycle, it may be that $LC' > ST$ such that LC' is much greater that ST; LC is often in the 100s in practice.

Agent speed is fixed at 2 ($AS = 2$). Rate of change of obstacles will be varied amongst 0, 2, 4, 6, 8 and 10 ($RC \in \{0, 2, 4, 6, 8, 10\}$). Therefore, $Dynamism \in \{0, 1, 2, 3, 4, 5\}$. The exact choice for RC (and thus for *Dynamism*) and the *fire:pack* ratio will be mentioned when applicable.

7.4 Naive Architectures for Base-lines

In order to have a base-line against which the performance of the new hybrid architecture can be compared, a simple or 'naive' architecture (called Naive-POP) was implemented. It has

no explicit intentions or desires as defined for the BDI model. The agent is provided with a single Golog controlling program and associated reward function. In this implementation, the program loops continuously over a nondeterministic action—nondeterministic between all available actions:

while *true* **do** [*sensloc* | *grab* | *eat* | *left* | *right* | *up* | *down* | *noop*].

Policies will always be sought with $h = 3$ and $c = 0.01$.[1]. Hence, after every three actions the agent performs, a new policy is generated with *BestDoPO* to depth 3, using the current belief state as input to *BestDoPO*.

The reward function is defined such that the agent gets a unit of reward for each cell directly next to or diagonally next to it containing a fire or power pack, a 100 units for eating a fire and 103 units for grabbing a pack. The idea is that the agent should move towards areas where there are more fires and power packs, but eating fire is a priority and grabbing packs even more so.

7.5 Full Observability vs. Partial Observability

Due to the obvious advantage of using the belief update function with the *reduction by probability cut-off* modification, this modified function will be used in all architectures and experiments in this and the next section.

In this section, we shall see the effects on agent performance when the agent has access to models of the world that assume (i) full or (ii) partial observability. Naive-POP is used for these experiments.

Two experiments will be conducted, one with an agent that is more accurate in its actions and observations, and one where the agent is less accurate. In both cases, when the agent intends performing *eat*, *grab*, *sensloc* or *noop*, the intended action is the outcome with 100% certainty.

Our agent that assumes partial observation has accurate models for actions and observations. Our agent that assumes full observation has an accurate model for actions, but it models observation as follows. The probability that the agent is in the location it thinks it is in, is 1.0, and zero for any other location.

For the experiments in this section, we chose to fix $RC = 2$, that is, *Dynamism* = 1. And for both experiments, 14 trials were performed with the *fire:pack* ratio being 9:27, 14 trials with 18:18, and 14 trials with 27:9; that is, 42 trials in an experiment batch.

[1] As a matter of interest, for the domain used in Section 7.2, the running time for generating a policy to depth 3 with $c = 0.01$ is on average, close to one second. Also note that although $c = 0.5$ resulted in an optimal policy (cf. Table 7.1) for the 5×5 world, $c = 0.01$ yielded good results for FireEater world in pre-experiment trials.

In the first experiment, the domain is less stochastic: When an agent's intended action is *left*, *right*, *up* or *down*, the probability that the outcome is the intended action, is 0.95, else it is 0.025 that it will move sideways—the agent never moves in the opposite direction to its intended direction.

When the agent observes its location (via *sensloc*), nature will let it perceive the location it *actually* is in with probability 0.96, else the agent perceives that it is in one of the eight neighboring cells with probability 0.005.

With this setup, the agent assuming full observation had $\mathcal{P} = 3.146$, and the agent assuming partial observation had $\mathcal{P} = 3.84$.

In the second experiment, the domain is more stochastic: When an agent's intended action is *left*, *right*, *up* or *down*, the probability that the outcome is the intended action, is 0.75, else it is 0.125 that it will move sideways.

When the agent observes its location (via *sensloc*), nature will let it perceive the location it *actually* is in with probability 0.76, else the agent perceives that it is in one of the eight neighboring cells with probability 0.03.

With this setup, the agent assuming full observation had $\mathcal{P} = 2.91$, and the agent assuming partial observation had $\mathcal{P} = 1.502$.

7.5.1 Analysis

In this environment and with this architecture, the agent assuming full observation fares worse in the less stochastic domain than the agent assuming partial observation. In contrast, the agent assuming full observation fares better in the more stochastic domain than the agent assuming partial observation.

These results go against intuition: in domains where observation is extremely noisy, one expects the performance of a planner that includes models of partial observability to be better than the performance of a planner that includes only models assuming full observability. Although the agent assuming partial observation outperforms the agent assuming full observation in the less stochastic domain, the opposite is true in the more stochastic domain. In the present case, the counter-intuitive results may be due to some detail in the implementation of the simulator (FireEater world) or in the naiveté of Naive-POP. To reverse these results is work for the future.

7.6 Comparing Naive-POP, BDI-POP and BDI-POP(R)

The three agents as implemented by the three architectures have identical knowledge bases, except for their programs and reward functions. That is, they believe the same actions are possible, with the same effects and associated probabilities. They both employ the exact same planner: *BestDoPO*. The models they have of their environment—with respect to actions outcomes and observational noise—are accurate:

The simulator will cause actions $left, right, up$ and *down* to be nondeterministic; the agent's intended action will be the outcome 95% of the time. The other actions will be deterministic. Observation will also be imperfect: the agent perceives its actual location with probability 0.96, else it perceives one of the eight neighboring cells with probability 0.005—as in Section 7.5.

In BDI-POP, we set $h^- = h - 1$—the search horizon that *Focus*$(\cdot)$ uses to determine program values. In BDI-POP(R), h^- is the search horizon that *Focus*$(\cdot)$ *and Reconsider*$(\cdot)$ use to determine program values. For BDI-POP(R) we also set $h^- = h - 1$.[2] Arguments F_V and F_I required by *Reconsider*$(\cdot)$ (cf. Section 5.2) are set as follows. $F_V = 0.5$, $F_I = 0.5$.

We shall investigate two agent specifications when considering the two BDI architectures. I shall identify two specifications: *behavior-set 1* and *behavior-set 2*. The only difference in the two specifications is in the desires, behaviors and reward functions of the agents.

Behavior-set 1 has the following specifications: There are four agent desires: $D = \{$`findFire`, `eating`, `findPowpac`, `grabbing`$\}$. `findFire` may be realized by two available behaviors, `eating` by one behavior, `findPowpac` by two, and `grabbing` by one. All behaviors have programs that are each a sequence of three actions. All behaviors include $left, right, up$ and *down* as action options. The '`findFire` behaviors' tell the agent to move towards regions where there are fires; one behavior includes sensing, the other not. The '`findPowpac` behaviors' are similar to the '`findFire` behaviors'. The behaviors that achieve `eating` and `grabbing` include the actions *eat* and respectively *grab* as options. There are six reward functions, one function tailored for each behavior.

The set of behaviors consists of:

- (`findFire1`, [*sensloc*; *sensloc* | $left$ | *right* | up | *down* | *noop*; *sensloc* | $left$ | *right* | up | *down* | *noop*], `findFire`, *nil*)
- (`findFire2`, [$left$ | *right* | up | *down* | *noop*; $left$ | *right* | up | *down* | *noop*; $left$ | *right* | up | *down* | *noop*], `findFire`, *nil*)
- (`eat1`, [*eat* | $left$ | *right* | up | *down* | *noop*; *eat* | $left$ | *right* | up | *down* | *noop*; *eat* | $left$ | *right* | up | *down* | *noop*], `eating`, *nil*)

[2] In pre-experiments, it was established that, in general, $h^- = h - 1$ dominates $h^- = h - 2$ in terms of the performance measure $\mathcal{P}$.

- (`findPowpac1`, [*sensloc*; *sensloc* | *left* | *right* | *up* | *down* | *noop*; *sensloc* | *left* | *right* | *up* | *down* | *noop*], `findPowpac`, *nil*)
- (`findPowpac2`, [*left* | *right* | *up* | *down* | *noop*; *left* | *right* | *up* | *down* | *noop*; *left* | *right* | *up* | *down* | *noop*], `findPowpac`, *nil*)
- (`grab1`, [*grab* | *left* | *right* | *up* | *down* | *noop*; *grab* | *left* | *right* | *up* | *down* | *noop*; *grab* | *left* | *right* | *up* | *down* | *noop*], `grabbing`, *nil*).

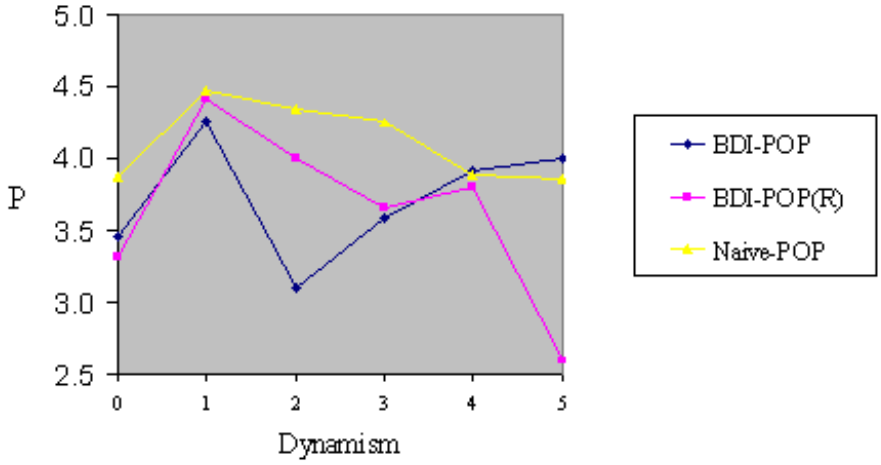

Figure 7.3: *Behavior-set 1*: Performance of Naive-POP, BDI-POP and BDI-POP(R) as *Dynamism* changes.

Behavior-set 2 has the following specifications:

Desires: D = {`findFire`, `findPowpac`}.

Behaviors: $\mathcal{B}$ = {(*findFireProg*, **while** *true* **do** [*eat* | *left* | *right* | *up* | *down* | *noop* | *sensloc*], `findFire`, *nil*), (*findPowpacProg*, **while** *true* **do** [*grab* | *left* | *right* | *up* | *down* | *noop* | *sensloc*], `findPowpac`, *nil*)}.

Reward functions: As required, there are two functions. They are both similar to the function described in Section 7.4 (cf. Appendix B).

The parameter for the number of obstacle changes per simulation cycle (*RC*) and the *fire:pack* ratio are the only parameters varied during experiments. Sixty trials per setting of *RC* were performed; (20 trials were performed with the *fire:pack* ratio being 9:27, 20 trials with 18:18, and 20 trials with 27:9).

The graphs in Figures 7.3 and 7.4 compare Naive-POP, BDI-POP and BDI-POP(R) for, respectively, *behavior-set 1* and *behavior-set 2* with respect to the performance measure $\mathcal{P}$ for varying dynamism of the world. (The result curve for the experiment on Naive-POP is the same in both figures—in Figure 7.3 till *Dynamism* = 5 and in Figure 7.4 till *Dynamism* = 4.)

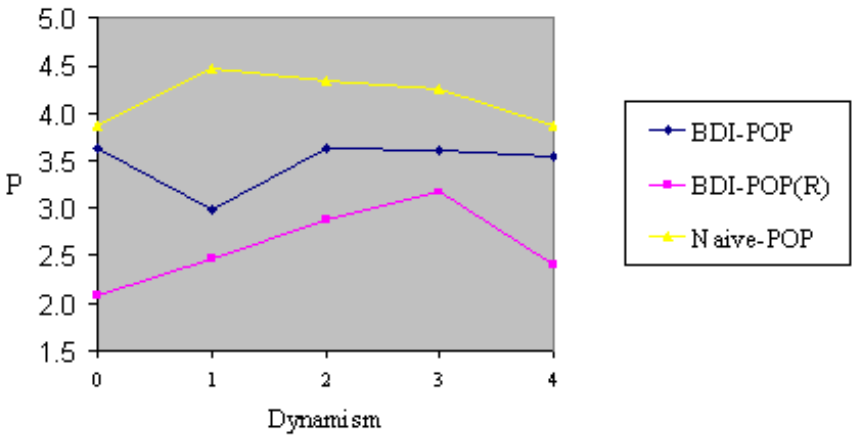

Figure 7.4: *Behavior-set 2*: Performance of Naive-POP, BDI-POP and BDI-POP(R) as *Dynamism* changes.

7.6.1 Analysis

In Figure 7.3 involving *behavior-set 1*, until level 4 *Dynamism*, the three architectures perform similarly, except for BDI-POP which has relatively bad performance at *Dynamism* = 2. When dynamism increases to level 5, the performance of BDI-POP(R) diverges dramatically for the worse. BDI-POP performs slightly better than Naive-POP at *Dynamism* = 5. Although reconsideration is supposed to enhance an agent's performance in more dynamic environments, it would seem that the current definition of the *Reconsider* function does not suit the FireEater testbed. Recall that the only difference between BDI-POP and BDI-POP(R) is the employment of reconsideration in BDI-POP(R); *Reconsider*(·) is thus most likely to blame for the weaker performance of BDI-POP(R).

Figure 7.4 clearly shows that Naive-POP dominates BDI-POP and BDI-POP(R) when *behavior-set 2* is used. Moreover, BDI-POP dominates BDI-POP(R) in this case. What these results show, is how the definition of the behavior (Golog programs) can influence the performance of the BDI-POP((R)) architectures.

In general, it is interesting that no matter the behavior set used, neither BDI-POP nor BDI-POP(R) dominates the Naive-POP architecture. It must be kept in mind though, that the comparison of the three architectures says nothing about the effectiveness of *BestDoPO*.

7.7 Discussion

In this chapter one sees that *BestDoPO* does work and it works for various scenarios.

Experiment 1 (cf. Section 7.2.1) illustrates nicely the value of having a planner that can generate policies for agents who maintain *belief states* (although the agent was unsure of where it was, it

could devise an optimal policy to get to the goal). Moreover, agents who model noisy sensors or partial observations, typically maintain belief states.

Also, we have seen that three of the optimizing methods applicable to *BestDoPO* make a significant positive difference, albeit for the simple domain setups of these experiments. What the results of Section 7.2 clearly illustrate, is that planning a long sequence of actions directly with a POMDP based planner is extremely inefficient. If an agent designer were interested in using *BestDoPO*, BDI-POP, BDI-POP(R), or even Naive-POP, he/she would be advised to select one of the optimization methods (and set its parameter, where applicable) such that it has the best performance for the task and environment being considered.

BDI-POP((R)) was developed and tested on FireEater world. The architecture was tested with various settings within the limits of FireEater world, however, tests should be conducted on other simulators or with physical robots to assess BDI-POP((R)) properly.

The following broad conclusion can be made: BDI-POP((R)) is approximately as effective as the Naive-POP architecture, in the framework of the FireEater world. More comments are made in the next chapter, about the results in this chapter, but there, an attempt is made to contextualize the results as related to the wider field of intelligent agents and robotics. Some *specific* issues are also discussed in the next chapter.

Chapter 8

Conclusion

8.1 Summary

We developed a Golog dialect called PODTGolog based on the situation calculus. PODTGolog is an extension of DTGolog, allowing for reasoning and planning in partially observable and nondeterministic domains, while providing the means to specify problems in a quantified predicate logic.

The PODTGolog 'interpreter' *BestDoPO* and the generic BDI architecture model were combined to take advantage of the benefits of each of the two frameworks. The resulting hybrid architecture is called BDI-POP, or BDI-POP(R) when it includes reconsideration (BDI-POP((R)) denotes BDI-POP and BDI-POP(R)).

In Chapter 7, we saw evidence for the potential of BDI-POP((R)). It did not perform as well as one might have expected, given the benefits that BDI theory is supposed to bring to an agent in complex and dynamical environments (given the performance measure $\mathcal{P}$). However, there is still much scope for improvement of BDI-POP((R)); see the *Discussion* section below.

What *has* been shown is that the proposed hybrid architecture is implementable; there is no obvious fundamental conflict in synthesizing our POMDP planner and the BDI model for agent control. The groundwork has thus been laid for the development of more sophisticated planning processes in the BDI-POP framework. Furthermore, PODTGolog is a powerful POMDP solver in itself; it comes with the expressive power of Golog programs, including the ability to express relations, complex world models, and executability, which can drastically reduce search space. We have also investigated several methods for optimizing *BestDoPO*. Some of the methods produced significant improvement in running time without affecting policy quality.

8.2 Research Questions Answered

Recall the thesis statement: *It is possible to define a belief-desire-intention architecture that employs a logic based planner that generates control policies for agents inhabiting partially observable stochastic domains, such that the performance of the agents is reasonable or such that the hybrid architecture shows* clear potential *for controlling agents with reasonable performance.*

The associated research questions are now answered:

1. **Q:** Can an existing language, DTGolog, be extended to generate policies for partially observable Markov decision process (POMDP) problems?
   A: Yes.

2. **Q:** Can a logic-based POMDP planner be integrated with the belief-desire-intention architecture? That is, is it possible to specify a 'reasonable' hybrid BDI/POMDP-planner architecture?
   A: Yes.

3. **Q:** Is there a performance gain in an agent when the agent is controlled by policies generated from the logic-based POMDP planner as compared to being controlled by policies generated from the logic-based planner assuming full observability?
   A: No, not with the particular domain problem and microworld used to answer the question.

4. **Q:** Is there a performance gain in an agent when the agent is controlled by the hybrid BDI/POMDP-planner architecture as compared to being controlled by a simpler architecture employing the new POMDP planner?
   A: No, with qualification: depending on the design of available behaviors (programs), BDI-POP and Naive-POP fare similarly or, Naive-POP dominates the others. This only holds for the particular experimental design.

8.3 Discussion

What remains unclear is how practical any architecture employing *BestDoPO* might be in realistically complex domains. With probabilistic outcomes and events in the world, the policy searches blow up very quickly with depth. For complete and optimal policies, POMDP solvers can deal with just a modest number of easily enumerated states. Policy trees of a fixed depth (as generated by *BestDoPO*) are not complete policies and thus less costly to generate. Realistically though, due to belief states being extremely numerous, the dimensionality of belief states due to the *situation equivalence problem* (cf. Section 6.2) in the situation-based representation

or simply due to the large numbers of conceivable states, *BestDoPO* makes POMDP planning intractable. Furthermore, the situation calculus, in principle, provides a good deal of expressivity (including quantified reasoning), which brings its own computational complexity issues (undecidability).

The methods and ideas discussed in Chapter 6 goes some way to alleviate intractability, but more is required. To further constrain this explosion in planning, especially in the domain of intelligent agents, it would be advisable to employ decision trees for policy search (instead of dynamic programming approaches) and keeping the horizon close, say $h = 3$ or $h = 4$. Although policy generation is required more frequently when smaller policies are sought, one gains very much in generation time. Such smaller, more frequently generated policies are then also more relevant at the moment of use. Planning then becomes more tractable and promotes reactivity in an autonomous system.

For a hybrid BDI/POMDP architecture to scale up to a domain more meaningful than a microdomain, the integration of more 'common sense' reasoning techniques into the architecture may have benefits. And the latest advances in POMDP solvers (for example, [105]) should be investigated for further ideas to improve the efficiency of *BestDoPO*.

A major concern for BDI-POP and even more so for BDI-POP(R), is that there are several parameters that can and must be set. Finding the best settings for these parameters is time consuming, and potentially, may consume other resources.

Where does BDI-POP((R)) fit in in the field of high-level control of agents and robots? Without empirical tests being performed with the architecture in real-world implementations, it is hard to answer this question. However, as discussed in the introduction, the architecture is meant for agents categorized as knowledge- and plan-based, especially those agents and robots operating in noisy and highly dynamical environments.

BDI-POP((R)) agents are neither purely decision-theoretic nor BDI-theoretic, but are combination decision/BDI-theoretic. BDI-POP((R)) would not form part of behavior-based or purely reactive architectures and most likely not of hierarchical architectures either. High-level control via BDI-POP((R)) may fit well into hybrid deliberative/reactive architectures. (Section A.5 in the appendix discusses the best known robot architectures.) The reader is invited to scan the contents of Chapter 3 for the company that BDI-POP((R)) would keep, especially ICL_{SC} [80], Bonet and Geffner's approach [10], ReadyLog [28] and all the BDI-based Architectures with Generative Planning.

The remarks at the end of the previous chapter (Section 7.7) are also relevant here.

8.4 Future Work

The relative sophistication of BDI-POP((R)) may not be applicable in very simple worlds such as FireEater. We would thus like to deploy our agents in larger, more complex worlds, with more complicated tasks for the agent to perform. This will also give more scope for the variety of programs that would be applicable, and the real power of the BDI model could come into play.

Because PODTGolog is based on the situation calculus, it is prone to the *situation equivalence problem* and the concomitant explosion of belief state dimension. Replacing PODTGolog in BDI-POP((R)) with a state-based language (for example, DyMoDeL [87]) would allow one to take advantage of methods for reducing this dimensionality problem. In particular, we have started research in this direction:

Consider an agent who maintains a belief state—a set of worlds that it believes it can possibly be in. Agents operating in partially observable domains must reason over belief states. If the agent's actions and observations are nondeterministic, the possible worlds in its belief state may keep increasing as it updates its beliefs.

The reasoning of an agent with belief states involves processes that consult the belief state. This may overtax the agent's mental capacity if the belief state representations are very large. In computational terms, the time and memory requirements of these processes is greatly influenced by the size (dimension) of belief state and the greater the opportunity for further explosion in the number of possible worlds as the belief state is updated. This has the consequence that the agent becomes less reactive in dynamic environments.

A method to heuristically limit the dimension of an agent's belief state is being considered. The method, at all times, limits the number of worlds in an agent's belief state to a designer supplied value (for example, 10). The challenge is to design the heuristics of the belief reduction method so as to lose a minimum of information during application of the method, by carefully selecting which worlds to discard.

Some work has been done that indicates experimentally and analytically that the belief reduction method we have in mind has potential benefits for agents who maintain and plan over belief states.

Fritz and McIlraith [34] have investigated combining non-Markovian qualitative (personal) preferences with qualitative decision theoretic programs. "The resultant DT-Golog program, maximizes the user's expected utility within the most qualitatively preferred plans," [34, p. 46]. It may be worthwhile to investigate how programs that incorporate personal preferences might restrict the deliberation process, perhaps making deliberation more efficient and agent behavior

more accurate.

More recent work by Khan and Lespérance investigates a more sophisticated approach to goal commitment. From their abstract:

> Most previous logical accounts of goals do not deal with prioritized goals and goal dynamics properly. Many are restricted to achievement goals. In this paper, we develop a logical account of goal change that addresses these deficiencies. In our account, we do not drop lower priority goals permanently when they become inconsistent with other goals and the agent's knowledge; rather, we make such goals inactive. We ensure that the agent's chosen goals/intentions are consistent with each other and the agent's knowledge. When the world changes, the agent recomputes her chosen goals and some inactive goals may become active again. [50]

Intentions may be represented as a structure of more and less abstract plans. Sohrabi, Baier and McIlraith have looked at HTNs to

> [...] address the problem of generating preferred plans by combining the procedural control knowledge specified by Hierarchical Task Networks (HTNs) with rich user preferences. ... To compute preferred HTN plans, we propose a branch-and-bound algorithm, together with a set of heuristics that, leveraging HTN structure, measure progress towards satisfaction of preferences. [99]

It may be worthwhile investigating the benefits of combining the work of Khan and Lespérance and of Sohrabi, Baier and McIlraith into a united framework. An idea is to supply the required preference information as POMDP utility information. The new framework would consist of HTN-like plans with values attached, combined with goal dynamics and priorities. Reasoning about goals within structured plans (networks) is already the approach of BDI theory. Combining the advances in separate works on HTNs and goal dynamics as reflected by the two papers mentioned just now [50, 99] may yield interesting (improved) agent reasoning.

There are plenty of opportunities for adding adaptation (learning) capabilities to agent reasoning systems. There are several elements of the agent architecture to apply adaptation techniques to. Consider the following.

One possibility would be to integrate Ivan Varzinczak's approach [106] to revising agents' action theories. From his abstract:

> Logical theories in reasoning about actions may also evolve, and knowledge engineers need revision tools to incorporate new incoming laws about the dynamic environment. We here fill this gap by providing an algorithmic approach for action theory revision. [106]

Another aspect of agent cognition that could benefit from adaptation, is in the trade-off between exploration and exploitation using reinforcement learning techniques [102]. In most domains of autonomous agents, the agents' main task is not only to learn, but to complete a task. However, it is usually desirable for an agent to learn something about its environments so as to complete its future tasks more efficiently. There is an inescapable trade-off in the resources for learning and adaptation (exploration) and task completion (exploitation). For example, a robot may have two routes it can take from the dining room to the kitchen while clearing the dinner table. It has only ever taken the one route and knows that this route is safe, but it may turn out that the other route is faster. But is the second route safe? The robot may waste time investigating the second route, but save time in the long-run if it finds that the second route is safe and faster than the first.

One aspect of agent cognition that would require learning is for the agent to learn the expected time it takes to perform an action, and with this knowledge, predict how long it has for doing deliberation before a new policy is required. For this last aspect, inspiration could be gleaned from work on anytime algorithms [89, 7, 74]. A robot may be willing to accept 'good enough' policies instead of only optimal policies, especially when action is more important than perfect plans. An example scenario is when a robot's master has ordered it to clear the dinner table in three or fewer minutes (because unexpected guests have arrived). The robot must just get all the dirty dishes to the kitchen, but does not have to pack the dishes perfectly neatly. In this scenario the robot must do some planning, but it is more important for it to do the job (more action) than to do the job precisely (suboptimal dish packing).

Lastly, there may be an opportunity to improve or extend work done which uses reinforcement learning—"applied to the first-order MDP representations induced by the program" [3]—to let an agent choose the optimal action when presented with a set of actions it may nondeterministically choose from. Finzi and Lukasiewicz [30] first applied reinforcement learning to first-order MDPs [12], and their work was extended by Daniel Beck and Gerhard Lakemeyer [3].

Appendix A

Paradigms and Theoretical Implementations for High-level Control of Agents and Robots

Consider the components that make up practical intelligence—the components in the mind of any person who can get things done effectively. As will be seen in this appendix, psychologists and roboticists have identified or named several more or less separate components of practical intelligence. There are hundreds if not thousands of ways to configure these components to form reasonable 'architectures' to intelligently control an artificial agent. In order to find good architectures for robots, roboticists have to keep matters relatively simple. Yet, striving to create intelligence, naturally makes matters complicated: for a machine to approach a semblance of real intelligence, its cognitive system must include certain fundamental components of intelligence, and these components must interact to form a synergetic system.

We shall look at the most prominent paradigms and architectures that have emerged from the robotics and agent community in their attempts to make robots and agents intelligent enough to perform tasks effectively and autonomously. The contents of this appendix are

- Section A.1: Definition of *Agent* and *Robot*.
- Section A.2: Intelligence.
- Section A.3: Plans.
- Section A.4: Levels of Control.
- Section A.5: Named Paradigms and Example Theoretical Implementations.

A.1 Definition of *Agent* and *Robot*

What makes an agent not 'merely' a sophisticated piece of software (for example, the newest interactive operating system)? What makes a robot not 'merely' a machine (for example, the newest PC or newest vehicle)? We begin with the definition of a robot and then use this definition to define an agent. The definition of what a robot is, is not standard. Here is one definition:

Robot

A *robot* is a physically embodied system with sensors and effectors, and a *robot* is autonomous, intelligent and motivated by intention. (See, for example, [6, 1]).

Even if this definition is agreed upon by a group, they may not agree on the meaning of the definition's individual sub-concepts.

Although the concept *robot* may be defined differently by different people, the definition given here will be used when referring to a robot in this dissertation.

The elements of the definition are now expanded on:

- *physically embodied system:* A machine with a function that is situated in the real world, affected by the forces in the environment, and exerting forces on objects in the environment.
- *sensor:* A device affected by some aspect(s) of the environment, this affect causing a measurement representing the affect made available to the rest of the system (of which the sensor is a part) for processing. The measurement is an electrical signal that the sensor transformed from some physical phenomenon (for example, pressure, heat, light, distance, sound). A sensor may perform an amount of preprocessing of the measurement data into a signal that is (more readily) usable by the higher-level processing systems.
- *effector:* A device that is *designed* to have an effect on the environment, or rather, objects in the environment. Because a robot is *situated* in its environment, it is an object in the environment, and therefore—if it simply moves a limb, for example—the robot thereby has an effect on the environment. The thing that caused the movement is thus an effector. A device that releases any kind of energy (into the environment) on receipt of a command signal (as designed) to do so, is an actuator. For example, a light that is mounted on a robot that can possibly be turned on by the robot's internal processes, is an effector. A motor is the classic effector, known technically as an actuator. A device attached to the robot but not controlled by the robot is not an effector for that robot. For example, a wrist watch is not an organ of the human body; the watch's modes are not directly controlled by the nervous system.
- *autonomous:* "Autonomy refers to systems capable of operating in the real-world envi-

ronment without any form of external control for extended periods of time," [6, p. 1]. Researchers of software agents ('bots') will disagree with this definition because it implies that autonomy is impossible for (intelligent) systems realized as software. According to Wooldridge, an autonomous agent has the locus of decision making concerning its actions; the locus is not outside of itself (possessed by another agent), and it has "its own beliefs, desires, and intentions, which are not subservient to those of other agents," [113, p. 3–4].

- *intelligent:* Intelligence in an environment is the ability to operate effectively on average (relative to the circumstances) in that environment to achieve an objective. The specification of 'effective operation' is expected to be supplied by the designer of the robot or agent to which the definition is being applied. For a more detailed exposition of intelligence, see the next section.
- *motivated:* Effection (activation of an effector) in order to perform some function or task, or to achieve some objective.
- *intention:* A signal or set of signals, or symbol or set of symbols representing signals, that are commands ready to trigger effection. "Motivated intention" could be called *proactiveness*, which Wooldridge [113] defines as exhibiting goal-directed behavior, moreover, a proactive agent with a goal or intention will try to *achieve* this goal or intention.

According to Bekey [6], a robot can sense, can act and can think (where "think" means: has subsystems that perform processes that may be thought of as cognition).

Agent

An *agent* may be a software 'bot', that is, an agent may have the same qualities of a robot, excepting that it is not embodied in hardware. Its sensors and effectors are *virtual*, for example, in object-oriented terms, sensors and effectors are actually inputs and outputs of the 'agent object'.

In this dissertation, the meaning of *agent* is (loosely) defined as in the preceding paragraph. Because robots and agents as just defined can both benefit from advances in cognitive robotics, no harm is done in referring to one and not the other when talking about concepts in cognitive robotics.

A.2 Intelligence

Intelligence is not easily defined and has not a generally accepted definition. However, (intelligent) people know intelligence when they see it, thus people usually simply refer to it when necessary, with the idea that their audience is intelligent. The definition of intelligence given

in the previous section is composed from my intuition and background knowledge. Meystel and Albus [69] mention that intelligence means different things to people in different fields of research; they do however give a definition of intelligence for the purpose of their book:

> *Intelligence* is the ability of a system to act appropriately in an uncertain environment, where an appropriate action is that which increases the probability of success, and success is the achievement of behavioral subgoals that support the system's ultimate goal. [69, p. 3]

To become intelligent, a human has to learn; in this sense, intelligence includes the ability to learn 'well': "The origin and function of intelligence is treated [in this book] from the standpoint of biological evolution, which is shown to be similar to the general process of learning," [69, p. 2]. Robots are *constructed* and all their initial intelligence is due to their (human) constructors. Therefore, initially—in the scope of the life of a robot—a robot's intelligence need not be judged on its ability to learn. However, for robots (and humans) operating in some environment for an extended period, if the robot continually operates ineffectively, we would say it is not learning well and is thus not intelligent. Hence, for agents operating in environments where adaptability is necessary for the 'effective' achievement of their objectives, the ability to learn (adapt) should be included in the definition of intelligence. On the other hand, if one judges an agent's intelligence over a short period, the inclusion of learning in the definition of intelligence is unnecessary.

If 'intelligence' is equated with the concept 'mind', then the following statement by Franklin [33, p. 412] is useful in our present exposition: "The overriding task of Mind is to produce the next action," where "produce" in this statement means to manage or create through a process. And this 'production' implicitly implies the effective achievement of an objective.

Meystel and Albus [69] depict "the elementary unit of self-organization," as in Figure A.1 which is a crucial component of intelligence, with three processes: *combinatorial search*, *focusing attention* and *grouping*.

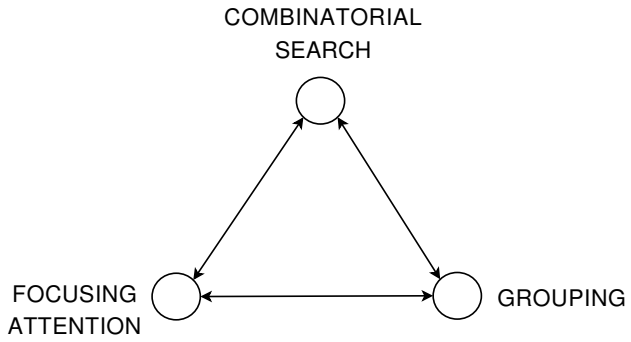

Figure A.1: The elementary unit of self-organization.

This unit of self-organization applies to data, information or knowledge at different levels of cognition.

An assertion that Meystel and Albus [69] make is that there are *degrees* of intelligence, and similarly, a major theme in Franklin's book [33] is that there are "degrees of mind". This is an important assertion, because it focuses our attention on the fact that the definition of intelligence is not dichotomous. That is, any definition of intelligence will (should) not be applicable as a decision process, that is, produce a definite positive or negative answer to the question of whether something is intelligent.

Franklin [33] mentions the "cognitive functions" as: *recognizing*, *categorizing*, *recalling*, *inferring* and *planning*. Other cognitive functions found in robotics are map building (environment/world modeling), performance monitoring, failure handling, projection (simulating events and action sequences), solving of unforeseen specific problems and learning. "In order to differentiate [...] more cognitive oriented functions from path planning, the term *deliberative* was coined," [71, p. 258].

To know which elements (cognitive functions) a truly intelligent robot should have, one might look at what psychologists say are the elements of intelligence in human beings. The topics (chapters) applicable to robotics covered in two text books on cognitive psychology [41, 101] are

- Perception
- Attention and Self-awareness
- Memory Processes
- Knowledge Representation, Retrieval, and Organization
- Visual Imagery
- Language/Communication
- Problem Solving and Creativity
- Reasoning and Decision Making

It is noteworthy that these two books hardly mention 'planning' as a topic in cognitive psychology (as an element of cognition). Ironically, in robotics, we think of planning almost synonymously with deliberation/cognition. Is this an indication that roboticists should consider changing their focus in robot deliberation away from planning?

A section is dedicated to planning nonetheless, because it is a central mechanism used by roboticists for the intelligent control of robots.

A.3 Plans

When we talk about planning in this section, we mean the reasoning done to find the most appropriate (effective or optimal) sequence of actions to perform to achieve an objective or sub-objective.

Planning may be part of decision-making, but it need not be. An example of when planning is combined with decision-making is when Markov decision process (MDP) techniques are employed for plan generation (for uncertain domains). Planning is always considered as one kind of *deliberation*. Plans could in a sense also fall under non-deliberative (even reactive) components of a robotic system, but we do not consider such 'low-level plans' in this dissertation. An atomic act generated in less than a few milliseconds or plans composed of atomic acts already existing as procedural knowledge, are examples of plans in low-level components.

Planning is either progressive (from the current situation to the goal) or regressive (from the goal to the current situation). Planning is either search-based (in a search space) or via constructive theorem proving (construction of the proof that the goal (statement) is achievable, results in a plan for achieving the goal) [88].

"In the plan-based approach, robots generate control actions by maintaining and executing a plan that is effective and has a high expected utility with respect to the robots' current goals and beliefs," [4, p. v]. Because plans in plan-based robots are the core element of control of a robot, these robots are as flexible as the plans they maintain; reasoning about, manipulating and adapting plans in these robots is thus equivalent to reasoning about, manipulating and adapting the robots' behavior.

Plan-based controllers need not *generate* plans, only utilize plans. Of course, they can also generate plans.

Beetz and Hofhauser [5] mention three advantages of plan-based control: (1) plans can contain the intentions of other agents, (2) experience can be compiled into plans and (3) plans can capture *strategic* considerations, that is, considerations involved in wider time horizons as opposed to situated (instantaneous) action selection.

One dimension of plans that most likely will influence an agent architecture, is whether plans are precompiled (written by hand and stored for retrieval as necessary), or generated online (automatically generated by a 'planning module' while the agent is performing its task). When plans are precompiled, there is an obvious lack in flexibility in the choice of plans; many situations may arise that the human plan-writer could not foresee and the agent would not have a plan for such situations. The benefit of having a library of ready-made plans is that the agent will not waste any time generating a plan. This is useful in time critical situations.

On the other hand, an agent that generates its own plans can theoretically always have a plan for

any situation. For this to be true in practice, the agent would need many and various sensors, and a complex planner. Even if the agent did have such a planner and all necessary sensors, complex planners take long to generate a plan. This introduces us to the classic dilemma in plan-based robotics: Good plans take long to generate, and while they are being generated, the world situation for which the plan is applicable will likely have changed (the real-world continually and continuously changes). There are three approaches to remedy the dilemma, with no one approach alone being sufficient (for real-world robotic applications): (1) Make the planner more sophisticated by applying clever approximations, heuristics and theoretical insights. (2) Have lower level control systems in the architecture to monitor plan validity, and replan only when necessary and only as much as necessary. (3) Let low level control systems recommend/cause appropriate actions when a plan is not appropriate (for example, *If a brick is flying towards your head, Then duck*).

A combination of both precompiled and generated plans may turn out to produce better performance of an agent in certain (most?) domains than either alone.

When a control architecture involves plans (is plan-based) the kind of plans used and the type of architecture are usually mutually influential. Some well known kinds of plans are linear, partial-order, conditional and hierarchical task network plans [88]. So, for example, an architecture using linear plans would typically have to periodically confirm the validity of the plan. Partial-order plans would be applicable to a robot that can perform some behaviors or actions simultaneously. An architecture employing conditional plans needs to make observations at certain points in the plan execution to correctly decide which subtree of the plan to execute next. In the case of hierarchical task network plans, the architecture of the robot will need knowledge about tasks that is organized hierarchically before planning (if plans are generated and not precompiled) and the system need to be able to execute hierarchical plans.

Another dimension is whether experience is applied to plans. In terms of precompiled plans, new plans—whole or partial—can be learned. In terms of generated plans, new features, rules, etcetera, may be learned or old ones modified through experience, which influences the plans generated in future.

Planning in the light of control for robots in an uncertain world has not yet been discussed. Especially in this dissertation, we need to cover planning for plans applicable to uncertainty in the effects of actions and uncertainty in the accuracy of readings of sensors. There are various models of uncertainty and algorithms for finding plans within those models. Value-iteration is a popular and well-understood technique for generating a control policy (universal plan) for (intelligent) systems operating in domains where acting and sensing involve uncertainty [104]. Other algorithms for planning in uncertain domains include "the A* algorithm, which uses a heuristic in the computation of the value function, or direct search techniques that identify a locally optimal policy through gradient descent," [104, pp. 508–509]. The planner developed

in this dissertation (Chapter 4) uses the latter technique, here called the *decision tree roll-back procedure*.

To end this section, approaches to the use of plans from four articles in the collection titled "Advances in Plan-Based Control of Robotic Agents" [4] are mentioned. The details are not important; it is only to give more concrete examples of the application of plans in robotics.

- Beetz and Hofhauser [5] use RPLplan, for precompiled plans and the learning of subplans;
- Burkhard *et al.* [18] use partial hierarchical plans—plans are "set up" or "prepared" ("built from options") in an option tree; this is closer to precompilation than generation of plans, and no learning is involved;
- Karlsson and Schiavinotto [49] use PTLplans, which generate conditional plans and can handle degrees of uncertainty, and no learning is used;
- Zilberstein *et al.* [116] use an MDP planner—two variations: (1) with choice between subplans with associated utilities, (2) with hierarchical reinforcement learning.

A.4 Levels of Control

A.4.1 Definitions of *Architecture*

This section begins with definitions of what a robot *architecture* is. "An architecture is a framework consisting of functional modules, interfaces, and data structures. A reference model architecture defines how the functional modules and data structures are integrated into subsystems and systems," [21, Section 17.3.1, p. 658]. For Bonasso, *robot architecture* means "the arrangement of control software for the robot," [9, p. 193]. For Gat [36], *architecture* in robotics means "a set of constraints on the structure of a software system," [36, p. 210]. Bekey [6, p. 98] defines *architecture* as "the practical structure of a robot's software [...] its goal is to define the way in which sensing, reasoning, and action are represented, organized, and interconnected." Lastly, Arkin [1, p. 125] defines *robot architecture* as "the discipline devoted to the design of highly specific and individual robots from a collection of common software building blocks."

In terms of the three levels of abstraction defined in the introductory chapter, an architecture is a *theoretical implementation*, but it is more generic that a *robot* architecture.

A.4.2 The Control Dimension

Bekey [6] points out that there are two levels of control in a robot. He clarifies the difference between the two levels with the example of the design of the control of a missile: there is control *of* a trajectory and control *about* a trajectory, corresponding to high-level control in robotics and respectively low-level control. "Control of the trajectory is more commonly known as *guidance*," [6, p. 98]. Control about the trajectory concerns issues of stability and oscillation. This two-level view of control thus excludes decision-making (of which trajectory to follow).

As an aside, an interesting connection between control and architecture is mentioned by Bekey [6] who quotes Meteric (1992a):

> An architecture provides a principled way of organizing a control system. However, in addition to providing structure, it imposes constraints on the way the control problem can be solved. [p. 99]

Ultimately, a robot must decide which of its available actions to take and when to take it. Decision-making in robotics refers to high-level reasoning or high-level deliberation [68, Section 12.2.4]; [88, Section 25.7]; [1, Ch. 6]). It is "high-level" because low-level control of actions concerns atomic acts and is representationless. Decision-making/deliberation is a process concerning abstract (symbolic) events and concepts, including complex actions, that is, actions composed of several atomic actions. An example of an atomic act would be "bend knee five degrees"; a complex action would be "move one meter towards right" which—for a legged robot—presumably includes several knee-bending atomic acts and some other actions too.

> There are many different forms of decision making that exist in the current literature. Popular techniques include computation-based closed loop control, cost-based search strategies, finite state machines (FSM), and rule-based systems. [68, p. 475]

In this dissertation, control includes decision-making, but decision-making does not include low-level control. The lower the level of control of a robot, the less complex the actions being controlled, that is, the less abstract and symbolic the actions being controlled. There is not a clear line to distinguish high-level control and low-level control.

One important dimension with which to analyze an architecture or part thereof is how reactive or deliberative the architecture is. When there is little processing of stored information for control, we say the system is (more) *reactive*. When information processing frequently takes place before an action can be performed, we say the system is (more) *deliberative*. Figure A.2, adapted from Arkin [1], shows the spectrum of robot control systems.

Arkin [1] points out that deliberative architectures rely on world models. All parts of a deliberative architecture depend on plans (and replanning). He mentions that representing world

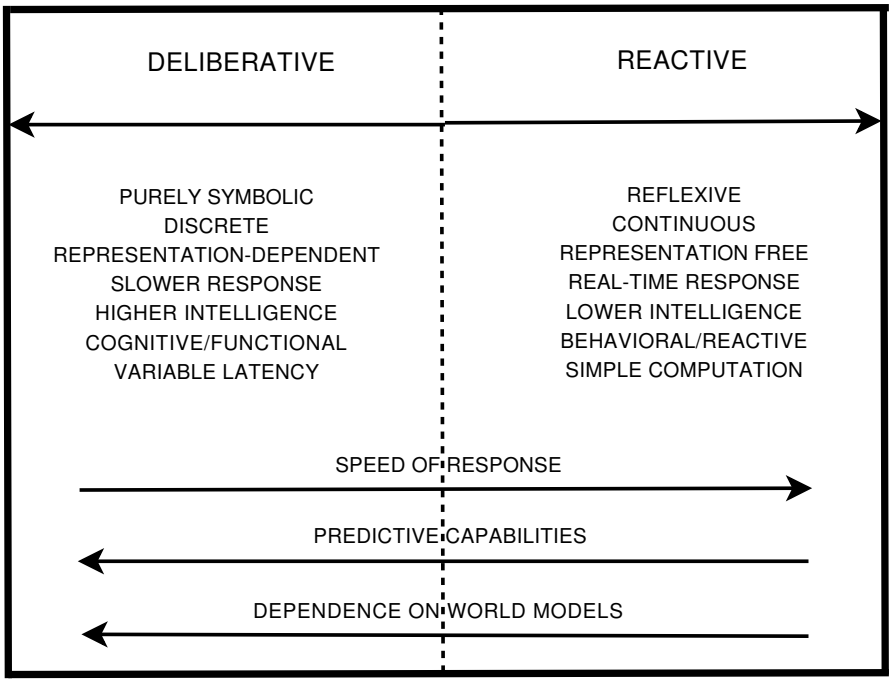

Figure A.2: The continuum of properties of deliberative and reactive robot architectures.

knowledge and having plans has its advantages, but constant (re)planning due to unforeseen circumstances or "gross assumptions about the world" can cause problems. This is the classic dilemma for plan-based robotics mentioned in Section A.3.

Reactive architectures have tightly coupled sensing and reacting. There is always an action 'recommended' for an environmental situation. Processing of plans and knowledge becomes unnecessary, and responsiveness to the environment thus becomes locally flexible and responses are in real-time [1]. On the down side of reactive systems, "the issues of action and perception are addressed, but cognition is ignored, often limiting these robots to mimicking low-level life forms," [1, p. 211–212].

The pros and cons of architectures at each of the two ends of the spectrum are listed in Table A.1.

Burkhard and colleagues [18] mention the possibility of classifying type of control via persistence of state: whether the agent (its architecture) maintains environment information about the past to inform future computations and whether the agent maintains information about past goals and plans to inform future commitment to goals and plans.

A.5 Named Paradigms and Example Theoretical Implementations

In this section we discuss the major paradigms for the high-level control of robots that have emerged over the past four or five decades, and some of their theoretical implementations.

Deliberative		Reactive	
Advantages	Disadvantages	Advantages	Disadvantages
- world model (state and dynamics) is useful	- information and plans become incorrect and invalid	- quick response is useful, especially in highly dynamic environments	- reasoning (deliberation) not possible: low intelligence systems
- through deliberation, actions can be better chosen	- deliberation takes long	- less chance for error because response is directly due to environment	- learning based on declarative memory is not possible
- future effects of actions can be considered	- architecture needs to be completed before testing can commence	- incremental/modular development is possible	- incremental design can cause overly complex systems

Table A.1: Pros and cons of architectures at the two extremes of the deliberation/reaction continuum.

A.5.1 Deliberative

Sense-Think-Act

One of the first paradigms for robot control, during the 60s and 70s, was based on Artificial Intelligence: the *Sense-Think-Act* (STA) or *Sense-Model-Plan-Act* cycle. Figure A.3 represents this approach. We shall refer to it as the *STA* architecture henceforth. It was the first structured method designed specifically for robotics. For example, refer to the work done at SRI International from the mid 60s to mid 70s [73].

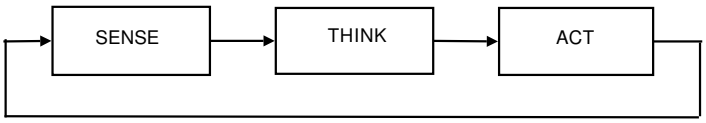

Figure A.3: The sense-think-act cycle.

Modeling and planning take much too long for robots with an architecture based on this paradigm to be useful [55, 36, 1]: the STA architecture lies far to the "Deliberative" end of the deliberative/reactive spectrum shown in Figure A.2 (Section A.4.2); as such, this architecture has all the disadvantages of a deliberative architecture to the worst degree: information and plans become incorrect and invalid by the time they are applied due to the latency between sensing and acting caused by thinking (see Table. A.1, Section A.4.2).

Arkin [1] equates traditional deliberative control with hierarchical control systems and Murphy [71] explicitly calls the STA architecture hierarchical. We believe it is more appropriate to label this traditional architecture based on STA *completely cyclic* rather than hierarchical. This is justified by the fact that it is the only architecture in which control flows through the *whole* system in cycles with a strict order of operation. (Hybrid architectures—covered later—may have cycles, but any cyclic operation is isolated to a layer, module or component. Also, as will

be seen in the next section on hierarchical systems, the characteristics of the STA architecture are different from those of hierarchical systems.)

Belief-Desire-Intention

Although the belief-desire-intention paradigm is not as abstract as the STA paradigm, in its general form it is a paradigm and not an architecture. There are many 'BDI architectures' which are instances of the BDI paradigm. IRMA is one such architecture [16].

The BDI paradigm is more complex than STA. As the name of the paradigm suggests, its primary components are beliefs, a set of desires, and a set of intentions. In this paradigm intentions are plans or goals that have been committed to. Desires are an agent's motivations, that is, what it wants to achieve. Intentions are a subset of its desires, the desires it has committed resources to.

The BDI paradigm and some derived architectures are discussed at length in Section 2.3.

Cognitive Psychology

Some agents are based directly on the psychological processes observed and theorized to occur in the human mind. *Soar* is one such architecture "useful for creating knowledge-rich agents that could generate diverse, intelligent behavior in complex, dynamic environments," [57, p. 224]. It is an architecture in pursuit of developing a system capable of general intelligence, not for solving specific problems with tailored algorithms [58].

Another architecture that falls in this category is OSCAR, "a fully implemented architecture for cognitive agents, based largely on the author's work in philosophy concerning epistemology and practical cognition," [79, p. 275 (abstract)].

Cognitive Robotics

Cognitive robotics is more concerned with high-level *robotic* control than with artificial intelligence for general problem solving. "In its most general form, we take cognitive robotics to be the study of the knowledge representation and reasoning problems faced by an autonomous robot (or agent) in a dynamic and incompletely known world," [62]. ReadyLog [28] is one programming language for cognitive robotics.

Discussion

Although the latter three deliberative paradigms are classified as deliberative, they are not purely deliberative in the sense that STA is. Because of their relative sophistication, taking the best elements of agent reasoning and control techniques from various sources, they are approaching being hybrid paradigms (cf. Section A.5.4). In fact, practical efficacy of implemented instance architectures of the latter three paradigm are comparable to other contemporary architectures' implementations. At the present time though, belief-desire-intention and cognitive psychology

paradigms are designed more for implementations of softbots, and cognitive robotics paradigms more for robotic agents.

A.5.2 Hierarchical

The traditional hierarchical robot architecture has three or more layers organized as hierarchies of control. Higher layers in the hierarchy produce or use plans that are more global and abstract in nature than at lower levels. Lower levels work at higher frequencies, that is, take care of finer-grained events that thus occur at more rapid frequencies. Communication and control flow strictly vertically to adjacent layers. Control may be data-driven, that is, driven by reactive requirements, or control may be goal-driven, that is, driven by reasoning about goal achievement [1, 71, 69]. In Figure A.4 the reader can see graphically the design of a hierarchical architecture.

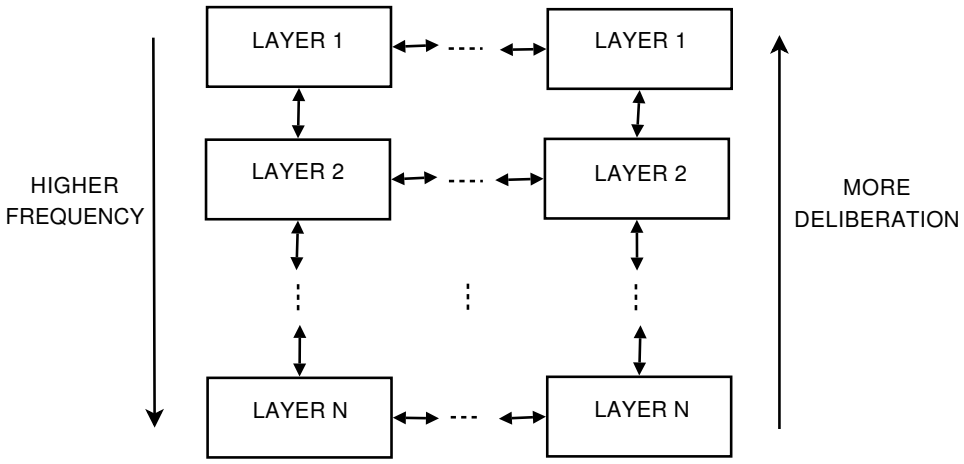

Figure A.4: The traditional hierarchical architecture.

The hierarchical architecture has its roots in the intelligent control community, and has "deliberative reasoning methods as its principle paradigm," [1, p. 21]. This architecture precedes the reactive, behavior-based architecture and was first applied to robotics (as opposed to non-robotic autonomous systems) from the 80s onward. "Hierarchical systems have largely fallen out of favor except for the NIST Realtime Control Architecture [RCS]," [71]. (See Section A.5.5; also see James Albus' curriculum vitae at http://www.isd.mel.nist.gov/personnel/albus/vitae.htm (retrieved: 12-11-2008).

There *do* still exist some advocates of hierarchical systems for robot control: Zilberstein *et al.* [116] describe the K9 rover software architecture, a four-layer hierarchical architecture with a decision-theoretic controller (planner) at the top of the hierarchy, which is the locus of control.

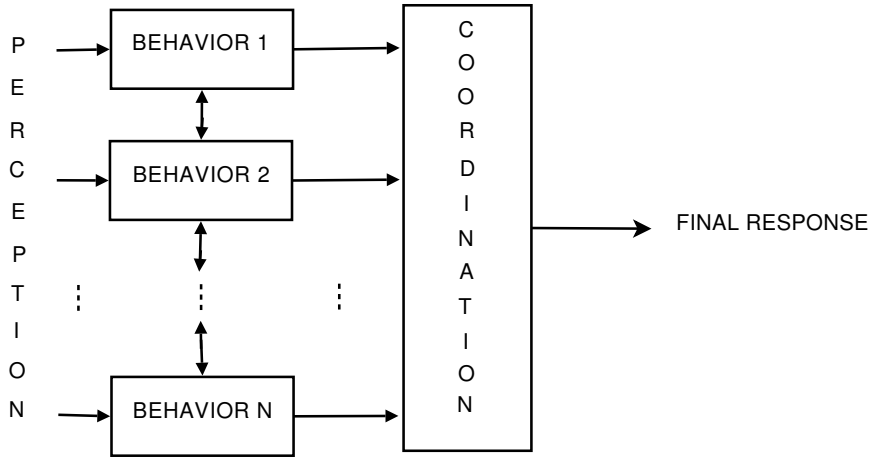

Figure A.5: The generic behavior-based architecture.

A.5.3 Behavior-Based (Reactive)

In the early 80s there was a paradigm shift towards reactive, behavior-based robotics, moving away from planning and the problems accompanying planning [55]. Robots started to move faster and then they actually became useful. Figure A.5 shows the basic scheme of a behavior-based robotic system.

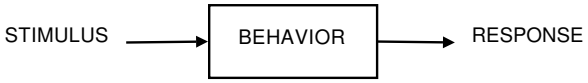

Figure A.6: The schema of a stimulus-response diagram.

We can understand a 'behavior' as a computational construct by looking at a stimulus-response diagram [1] (Figure A.6):

A behavior as a construct in robotics is also called a 'skill' to differentiate it from a psychological behavior.

From this perspective a 'behavior' is a construct that transforms a stimulus into a response, or that modulates some response according to some stimulus. Given a set of possible stimuli and a set of possible responses, a set of behaviors could be applied to link them. A behavior-based-robot designer would then choose a method to combine all the behaviors. That is, all the responses of the behaviors need to be coordinated. Arkin [1] discusses two approaches to *behavior coordination*: (1) competitive, where one out of two or more conflicting behaviors will 'win' according to some arbitration rules; the 'losing' behaviors being suppressed, and (2) cooperative, where the responses of 'conflicting' behaviors are fused or mixed to form a new amalgamated response.

A behavior-based architecture can be purely reactive or not purely reactive, nevertheless reactive. In other words, knowledge may be added to a reactive system such that the system is no longer *purely* reactive, yet is still behavior-based. Although behavior-based methods "pro-

vide excellent responsiveness in dynamic environments," [1, p. 173], many roboticists feel that adding an amount of representational knowledge has its place in such robotics.

Furthermore, Arkin [1] mentions that some roboticists believe that behavior-based robotics is limitless with respect to the design of intelligent systems, while others are concerned that it cannot scale up to human-level intelligence. Currently though, the robotics community has more perspective on the debate about the efficacy of the behavioral approach: we are beginning to agree that the behavior-based approach will not scale up and that it need not, because it has its application in lower levels of a larger architecture.

One architecture that is based on behavior is the *subsumption architecture*. The most important problem found by Arkin [36] and that may occur for other behavior-based architectures "is that it is not sufficiently modular [...] Because upper layers interfere with the internal functions of lower-layer behaviors, they cannot be designed independently and become increasingly complex," [36, p. 196].

Most robot designers using only reactive behaviors found that it was more an art than a science to get the robot to behave correctly overall. Some designers were asking whether the arrangement of the set of skills could be structured dynamically—structured to tackle one task, and then restructured to tackle another task [71]. Such real-time structuring would require plans! Enter the hybrid deliberative/reactive approach (covered next).

A.5.4 Hybrid Deliberative/Reactive

From around the late 80s and early 90s there was a return to planning:

> We might call this approach *P-SA* [T-SA in our terms]; that is, the robot plans based on initial conditions and common knowledge (P) and then executes this plan using sense-act (SA) behaviors, replanning only when the reactive behaviors run out of routine solutions. [55, p. 12]

The P-SA (T-SA) approach is the *hybrid deliberative/reactive* architecture.

Because each of the two paradigms is understood best while operating on its own as an independent system, most hybrid systems provide a clear demarcation between them [27, p. 164–165].

A hybrid deliberative/reactive architecture (HDRA) according to Arkin [1], attempts to combine the deliberative aspects of the completely cyclic and hierarchical architectures with the reactiveness of behavior-based architectures. The HDRA keeps the best features of both paradigms, letting features of one paradigm compensate for lacking beneficial features of the other. Dudek and Jenkin [27] say that neither a completely behavioral nor completely deliberative control

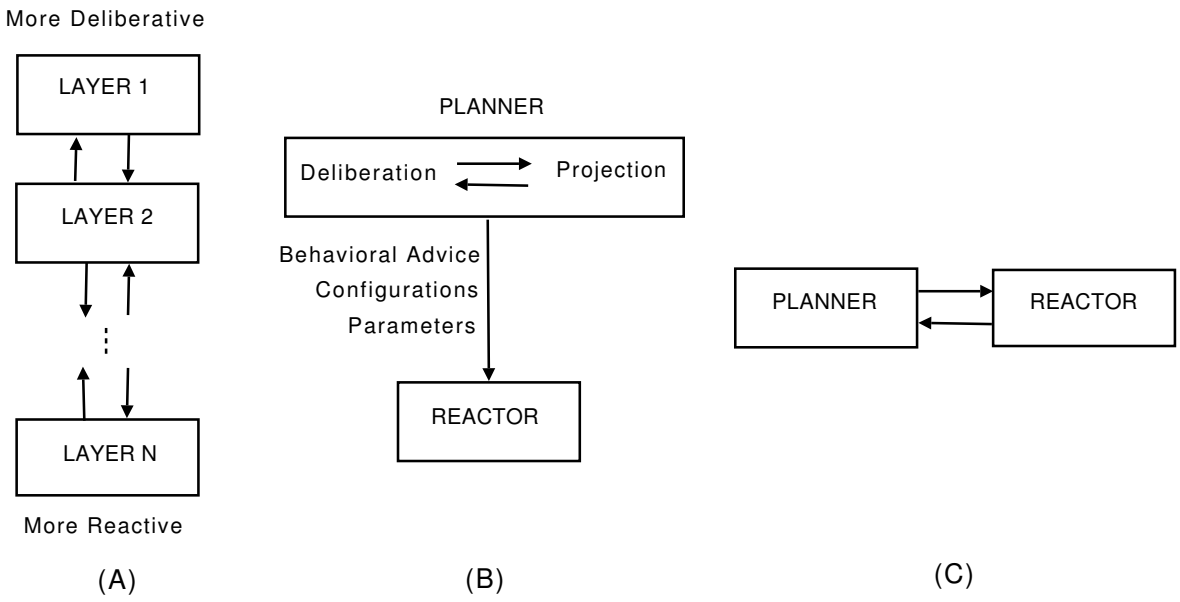

Figure A.7: Typical deliberative/reactive control strategies.

paradigm is appropriate for most robotic tasks. Figure A.7 represents three HDRAs [1], that is, three typical designs in which deliberation and reaction can be tied.

The three designs are

- *Hierarchical integration* (A), with higher levels being more deliberative and lower levels more reactive. This differs from the traditional hierarchical architecture in that in the hybrid version, different levels are more isolated, each level a functional unit using distinct types of knowledge representation and reasoning [1].
- *Planning to guide reaction* (B), that is, plans are used for advice; the reactive system could control the robot without the deliberative component.
- *Coupled planning-reacting* (C), where planning and reaction are concurrent activities, each affecting the other.

Murphy [71] also categorizes hybrid architectures into three categories:

- *Managerial*, with the manager at the top level and subordinates lower down (authority decreasing with lower levels). One level can only direct or modify the level directly below it. A property of this architecture is that the agent *fails upwards*, that is, if an operation at some level fails, the level directly above the failing level must attempt to rectify the problem, else pass the problem on upwards. This architecture is comparable to *hierarchical integration*.
- *State-hierarchy* organizes activity by "scope of time knowledge", that is, levels or layers are grouped and organized by the time-scope associated with the concepts that a level operates over. This architecture is comparable to *planning to guide reaction*.

- *Model-oriented* is more top-down in flavor than the other two, and the central element in the architecture is the model of the environment: "This global world model also serves to supply percepts to the behaviors (or behavior equivalents). In this case, the global world model serves as a virtual sensor," [71, p. 278]. This architecture is comparable to *coupled planning-reacting*.

From a comment made by Arkin [36] about the three-layer architecture—to be described in the next section—we may glean a further dimension of hybrid architectures: the Deliberator can respond to *requests* for plans from the Executor or it can *supply* the Executive with plans that the robot must execute.

From the preceding discussions, one might categorize hybrid architectures in another way, by these two dimensions: the number of layers and whether deliberation or reaction dominates control. For example, planning to guide reaction is a two or three layer architecture where the reactive *or* deliberative component may dominate control. The same is true for coupled planning-reacting. And in hierarchical integration, deliberation would typically dominate (but reaction could too), and the number of layers is three or more; a hierarchical design with two layers is simply a planning-to-guide-reaction design (not a coupled-planning-reacting design).

Arkin [1] points out that the *interface* between the deliberative component and the reactive component of an HDRA is the most important and complex aspect of the HDRA.

Three-Layer Hybrid Architectures

"[...] the three-layer architecture [...] has now become the de facto standard," [36, p. 198]. "By far the most popular hybrid architecture is the three-layer architecture, which consists of a reactive layer, an executive layer, and a deliberate layer," [88, p. 933].

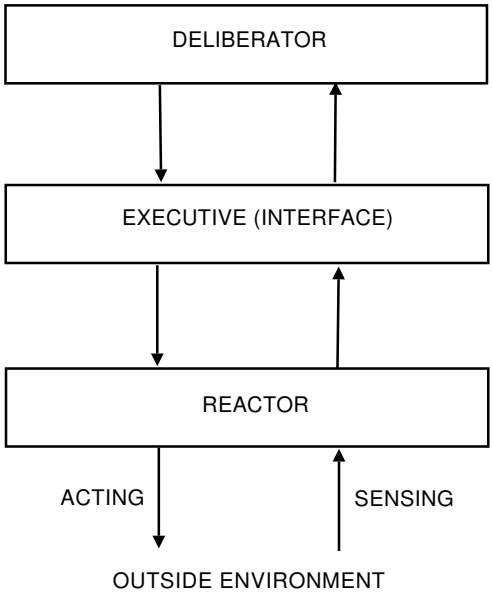

Figure A.8: The paradigm of a three-layer architecture.

This architecture has three distinct components, a deliberative component, a reactive component and an interfacing or executive component that links the other two. The three components have been named variously in different three-layer architectures. In this dissertation we shall refer to the top/deliberative layer as the *Deliberator*, the middle/interfacing/executive layer as the *Executive* and the bottom/reactive layer as the *Reactor* (Figure A.8).

The planning-to-guide-reaction and the coupled-planning-reacting designs may each be a three layer architecture if they include an interface component. The hierarchical-integration architecture cannot be a three layer architecture, because even if it is a three layer hierarchy, the middle layer is not strictly an interfacing or executive component.

According to Gat [36], the three-layer architecture is not due to theoretical considerations, but rather to empirical observations of the behavior of robotic systems in their environment. Nonetheless, a theoretical explanation for the emergence of the three-layer architecture as one of the dominating architectures for robots, is that each of the layers can be associated with the status of the use of representation (or state): In idealized terms, the Deliberator has state concerning future events, the Executive has state concerning past events, and the Reactor has no state.

In a three-layer architecture, Gat points out, plans are not executed in the traditional way: "An approach called *conditional sequencing* [...] is a more complex model of plan execution motivated by human instruction following," [36, p. 202]. Because a conditional sequencing system has control constructs more complex than provided by traditional programming languages, the system is constructed with "a special-purpose language like RAPs (Firby 1989), PRS (Georgeff 1987), the Behavior Language (Brooks 1989), REX/GAPPS (Kaelbling 1987, Kaelbling 1989, Bonasso 1992), or ESL (g98 1997)," [36, p. 202].

Bonasso and colleagues [8] describe the 3T architecture and compare it to the architectures and languages ATLANTIS, Cypress, InteRRaP and SIM_AGENT.

Kim, Shin and Choi [51] present a "plan-based control architecture for intelligent robotic agents" with three layers: *deliberative*, *sequencing* and *reactive* layers. The deliberative layer is implemented with UM-PRS [60] as a kind of plan executive employing BDI theory: it is "composed of five primary components: a *world model*, a *goal set*, a *plan library*, an *interpreter*, and an *intention structure*," [51, p. 562].

Other Hybrid Architectures

Phoenix [17, p. 332] is a three level hybrid architecture that has the hierarchical integration or managerial design. It was developed for student research in shallow-water sensing and control for an autonomous underwater vehicle. *Phoenix* has a 'Strategic Level' at the top, a 'Tactical Level' below that, and an 'Executive Level' at the bottom. The 'middle' (tactical) level is not explicitly an interface nor an executive component.

Burkhard *et al.* [18] developed the *Double Pass Architecture*, which uses case-based reasoning and which can switch high-level behavior—which goal to seek—in real-time. They point out that the hybrid deliberative/reactive architecture (HDRA) is not appropriate for domains with "total dynamics" (changing high-level goals) as opposed to "local dynamics" (high-level goals remain fixed). The deliberative component of a HDRA may run into reasoning bounds; the double pass architecture implements concepts of bounded reasoning to deal with said bounds.

The *Animate Agent architecture* is a two-layer HDRA [31]. One layer is the "Skill Level", the other is the "Task Execution Level". It is behavior-based at the skill level and has "sketchy plans" at the task execution level where the plans are broken down into steps that the skills can achieve.

Another two-layer architecture is that of Schönherr and Hertzberg [94]. Their architecture is called DD&P. It comprises "a deliberative and a behavior-based part as two peer modules with no hierarchy among these two parts," [94, p. 249]. Interaction between the two modules is regulated by flow of information similar to that of the STA architecture. They highlight two aspects of their architecture: their implementation of the 'plans-as-advice' approach, and how the symbols in the plans are grounded via "chronicle recognition".

Saphira [54] is a HDRA. It is behavior-based at lower levels of cognition (at reactive levels). A plan library and other (more sophisticated) modules make up the deliberative 'layer'. There is a control and coordination system present (PRS-Lite) (cf. Section 2.3.4), but it is not clearly an interfacing/executive layer. Saphira is thus not a three-layer architecture. "The organization is partly vertical and partly horizontal," [54, p. 219]. Murphy [71] categorizes Saphira as a model-oriented architecture.

A.5.5 The Reference Architecture for Intelligent Systems

The *Reference Architecture*, due to Albus and Meystel [69, 21], deserves some attention as a potentially important architecture for autonomous intelligent robots. The organization of the architecture is both hierarchical and multiresolutional, becoming finer grained at lower levels. Each level contains a number of 'subagents', 'computational nodes' or "elementary functioning loops" (ELFs) consisting of four modules ("elements of intelligence"): modules for sensory processing, world modeling (including a knowledge base), behavior generation and value judgment. There may be several or several hundred of these ELFs in a level.

The longest-term planning (for example, planning for twenty-four hours) happens at the highest level. Reflexiveness (via 'reflexive arcs') is accounted for as part of 'instinct' in general. "Innate reflexes can and should be built-in within the intelligent system such as autonomous mobile robots just at the stage of design," [69, p. 189].

One figure "illustrates the temporal flow of activity in the task decomposition of sensory-processing systems" [69, Figure 4.5, p. 169] and has at the bottom level, a three millisecond time interval "chosen as the shortest servo update rate because that is adequate to reproduce the highest bandwidth reflex arc in the human body," [69, p. 166]. The reference architecture thus has a definite reactive/behavioral element to it.

The 4D/RCS architecture is one realization of the reference architecture, that has been successfully implemented on several robotic platforms [69, 21]. (4D denotes a particular machine vision system used within the realization, and RCS stands for *Real-time Control System* developed at the National Institute of Standards and Technology, USA)

A.5.6 One Useful Classification of Architectures

Although quite outdated by now, Müller [70] provides a classification and taxonomy of architectures based on their class of application. He also discusses a set of guidelines for choosing an agent architecture for an application. As a matter of interest, Müller classifies the following as *deliberative agents*: IRMA, PRS, dMARS, SOAR, Cypress and Agent0 / PLACA, and the following as *layered approaches*: RAPs, ATLANTIS, 3T, Lyons & Hendriks, TouringMachines, InteRRaP, SIM_AGENT and NMRA. Please consult Müller [70] for details.

Appendix B

Source Code

The source code for BDI-POP (not BDI-POP(R)) is given here, including the definition of *BestDOPO*. It is written in ECL^iPS^e (Prolog) (an open-source software system available from http://87.230.22.228/) The implementation of the BDI controller and the simulator are not modular, that is, they do not form separate modules but are intermixed. *BestDOPO* is seperately specified though. Some predicates are not given here (to save space), but they are mentioned ('...' indicates that the details of a predicate are omitted). The problem description is provided as only one example of how to implement a POMDP-based agent (for FireEater world).

```
:- dynamic(simAtt/1).
:- dynamic(agAtt/2).
:- dynamic(obstacleList/1).
:- dynamic(FiresList/1).
:- dynamic(futureFiresList/2).
:- dynamic(powpacList/1).
:- dynamic(futurePowPacList/2).
:- dynamic(initAgentPos/1).
:- set_flag(print_depth,500).
:- pragma(debug).

:- lib(cio).
:- lib(ic).

:- op(800, xfy, [&]).   /* Conjunction */
:- op(850, xfy, [v]).   /* Disjunction */
:- op(870, xfy, [=>]).  /* Implication */
:- op(880, xfy, [<=>]). /* Equivalence */
:- op(950, xfy, [:]).   /* Action sequence */
:- op(960, xfy, [#]).   /* Nondeterministic action choice */
```

```
/* ---- BDI-POP: ---- */

/* ---- An integration of POMDP theory into a Belief-Desire-Intention agent model
        This implementation in ECLiPSe Prolog by Gavin B. Rens, November, 2009
        Meraka Institute, South Africa, & University of South Africa      ---- */


/* ---- PODTGolog: ---- */

/* ---- A POMDP planner developed by Gavin B. Rens ---- */
/* ---- PODTGolog is an extension of DTGolog (an MDP planner)
        Retrieved 4 September, 2008 from
        http://www.cs.ryerson.ca/~mes/publications/appendix/
        (appendix to M. Soutchanski's PhD dissertation)       ---- */



/* ---- PODTGolog Interpreter ---- */


/* Null action
*/
bestDoPo(nil : E,R,B,H,Pol,V,Prob) :- H > 0,
   bestDoPo(E,R,B,H,Pol,V,Prob).


/* Stop action
*/
bestDoPo(stop : _,R,B,_,Pol,V,Prob) :-
   Pol=stop, rewardBelief(R,V,B), (Prob is 0.0).


/* Zero horizon
*/
bestDoPo(P,R,B,H,Pol,V,Prob) :- H =:= 0,
   Pol=stop, rewardBelief(R,V,B), (Prob is 1.0).


/* Test action
*/
bestDoPo(?(C) : E,R,B,H,Pol,V,Prob) :- H > 0,
   (holdsBelief(C,Degree,B) ->
      bestDoPo(E,R,B,H,Pol,V,Prob1)),
      Prob is Prob1*Degree, !
      ;
      (Pol = stop, V is 0, Prob is 0.0).
```

```
/* Nondeterministic choice of actions
*/
bestDoPo((E1 # E2) : E,Rn,B,H,Pol,V,Prob) :- H > 0,
   bestDoPo(E1 : E,Rn,B,H,Pol1,V1,Prob1),
   bestDoPo(E2 : E,Rn,B,H,Pol2,V2,Prob2),
   (lesseq(V1,Prob1,V2,Prob2), Pol=Pol2, Prob=Prob2, V=V2 ;
   greatereq(V1,Prob1,V2,Prob2), Pol=Pol1, Prob=Prob1, V=V1).


/* Conditional statement
*/
bestDoPo(if(C,E1,E2) : E,R,B,H,Pol,V,Prob) :- H > 0,
   holdsBelief(C,Degree,B) ->
      (bestDoPo(E1 : E,R,B,H,Pol,V,Prob1),Prob is Prob1*Degree)
      ;
      bestDoPo(E2 : E,R,B,H,Pol,V,Prob).


/* Conditional iteration

   (Note that horizon has presendence over iteration, i.e.,
   even if the stopping criterion (C) is still satisfied,
   if H is zero, planning stops.)
*/
bestDoPo(while(C,E1) : E,R,B,H,Pol,V,Prob) :- H > 0,
   (holdsBelief(-C,Degree,B) ->
      bestDoPo(E,R,B,H,Pol,V,Prob)
      ;
      holdsBelief(C,Degree,B),
      bestDoPo(E1 : while(C,E1) : E,R,B,H,Pol,V,Prob1),
      Prob is Prob1*Degree ).


/* Sequencial composition
*/
bestDoPo((E1 : E2) : E,R,B,H,Pol,V,Prob) :-  H > 0,
   bestDoPo(E1 : (E2 : E),R,B,H,Pol,V,Prob).


/* Probabilistic observation
*/
bestDoPo(A : E,Rn,B,H,Pol,V,Prob) :-  H > 0, agentAction(A),
   (not believedPossAct(A,B), !,
      Pol = stop, V is 0, Prob is 0.0
      ;
      choiceObs(A,ChoiceObsList), !,
```

```
      bestDoObs(ChoiceObsList,A,E,Rn,B,H,RestPol,VF,Prob),
      rewardBelief(Rn,R,B),
      V is R + VF,
      Pol = (A : RestPol)
      ).


/* Observations possible (last (Ok))
*/
bestDoObs([Ok],A,E,R,B,H,Pol,V,Prob) :- H > 0,
   (not believedPossObs(Ok,A,B), !,
      Pol = stop, V is 0, Prob is 0.0
      ;
      beliefUpdate(Ok,A,B1,B),
      Hor is H - 1,
      bestDoPo(E,R,B1,Hor,Pol1,V1,Prob1), !,
      Pol = ( ?(observationIs(Ok),Pol1) ),
      probNebs(Ok,A,B,Pbs),
      V is V1*Pbs,
      Prob is Prob1*Pbs
      ).


/* Observations possible (>=2)
*/
bestDoObs([O1 | OtherOutcomes],A,E,R,B,H,Pol,V,Prob) :- H > 0,
   OtherOutcomes = [_|_],  % there is at least one other outcome
   (not believedPossObs(O1,A,B), !,
      bestDoObs(OtherOutcomes,A,E,R,B,H,Pol,V,Prob)
      ;
      bestDoObs(OtherOutcomes,A,E,R,B,H,PolT,VT,ProbT), !,
      beliefUpdate(O1,A,B1,B),
      Hor is H - 1,
      bestDoPo(E,R,B1,Hor,Pol1,V1,Prob1), !,
      Pol = (
      if(   observationIs(O1),  % then
            Pol1,  % else
            PolT)
      ),
      probNebs(O1,A,B,Pbs),
      V is VT + V1*Pbs,
      Prob is ProbT + Prob1*Pbs
      ).
```

```
/* ---- Some important predicates mentioned in the interpreter ---- */


/* For each possible observation, we specify when that observation is
   possible (perceivable), given the agent's belief.
*/
believedPossObs(O,A,[(S,_)]) :-
   possObs(O,A,S).
believedPossObs(O,A,[Head | Tail]) :-
   Head = (S,_),
   possObs(O,A,S)
   ;
   believedPossObs(O,A,Tail).


/* For each possible action, we specify when that action is
   possible (executable), given the agent's belief.
*/
believedPossAct(A,[(S,_)]) :-
   possAct(A,S).
believedPossAct(A,[Head | Tail]) :-
   Head = (S,_),
   possAct(A,S)
   ;
   believedPossAct(A,Tail).


/* Reward function over belief states
*/
rewardBelief(RewardFuncName,Rb,B) :-
   rewardBeliefAux(RewardFuncName,Rb,B).
rewardBeliefAux(_,0,[]).
rewardBeliefAux(RewardFuncName,Rb,[(S,P)]) :-
   reward(RewardFuncName,Rs,S),
   Rb is P*Rs.
rewardBeliefAux(RewardFuncName,Rb,B) :-
   B = [(S,P)|T],
   reward(RewardFuncName,Rs,S),
   Product is P*Rs,
   rewardBelief(RewardFuncName,Carry,T),
   Rb is Carry+Product.
```

```
/* Belief update function (state estimation function)
   beliefUpdate(Observation,Action,NewBelief,OldBelief).

   Using the 'belief state reduction by probability cut-off' method.
*/
beliefUpdate(O,A,B_out,B_in) :-
   beliefUpdateAux(O,A,B_temp1,B_in),
   findall((S1,P1), (member((S1,P1),B_temp1),P1 =\= 0.0), B_temp2),
   normalize(B_temp2,B_temp3),
   findall((S2,P2), (member((S2,P2),B_temp3),P2 > 0.01), B_temp4),
   normalize(B_temp4,B_out). %%% P2 > ??? must be empirically chosen

beliefUpdateAux(O,A,B_temp,[(S,P)]) :-
   findall((Sp,Pp),
      (Sp = doo(N,S),choiceNat(N,A,S),possAct(N,S),
         probObs(O,A,ProbObs,Sp), probNat(N,A,ProbNat,S),
         Pp is P*ProbObs*ProbNat),
         B_temp).
beliefUpdateAux(O,A,B_temp,[(S,P)|T]) :-
   findall((Sp,Pp),
      (Sp = doo(N,S),choiceNat(N,A,S),possAct(N,S),
         probObs(O,A,ProbObs,Sp), probNat(N,A,ProbNat,S),
         Pp is P*ProbObs*ProbNat),
         B2),
   beliefUpdateAux(O,A,B3,T),
   concat_list(B2,B3,B_temp).

normalize(B_temp,B_norm) :-
   sum_probabilities(B_temp, Denominator),
   findall((Sn,Pn),
      (member((St,Pt),B_temp), Sn = St, Pn is Pt/Denominator),
      B_norm).


/* The probability ProbNebs of reaching the new belief state,
   given the old belief state B, via O and A.
*/
probNebs(O,A,[],0.0).
probNebs(O,A,B,ProbNebs) :-
beliefUpdateAux(O,A,B,B_temp),
sum_probabilities(B_temp,ProbNebs).
```

```
/* ---- Some useful predicates mentioned in the interpreter ---- */


belief(_,0.0,[]).
belief(Q,Degree,[(S,P)]) :-
   holds(Q,S), !, Degree is P
   ;
   Degree is 0.0.
belief(Q,Degree,[(S,P)|T]) :-
   belief(Q,DegreeSoFar,T),
   (holds(Q,S), !, Degree is P + DegreeSoFar
   ;
   Degree is DegreeSoFar).


lesseq(V1,Prob1,V2,Prob2) :-
   Pr1 is float(Prob1), (Pr1 = 0.0),
   Pr2 is float(Prob2),
   ( (Pr2 \= 0.0)
   ;
   (Pr2 = 0.0) ,  V1 =< V2).
lesseq(V1,Prob1,V2,Prob2) :-
   (Prob1 \= 0.0) , (Prob2 \= 0.0) , V1 =< V2.

greatereq(V1,Prob1,V2,Prob2) :-
   (Prob1 \= 0.0) , (Prob2 = 0.0).
greatereq(V1,Prob1,V2,Prob2) :-
   (Prob1 \= 0.0) , (Prob2 \= 0.0) , V2 =< V1.


/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New.
*/
sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2), T2 =..[F|L2].

sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).


/* The holds predicate implements the revised Lloyd-Topor
   transformations on test conditions.
*/
holds(P & Q,S) :- holds(P,S), holds(Q,S).
holds(P v Q,S) :- holds(P,S); holds(Q,S).
holds(P => Q,S) :- holds(-P v Q,S).
holds(P <=> Q,S) :- holds((P => Q) & (Q => P),S).
```

```
holds(-(-P),S) :- holds(P,S).
holds(-(P & Q),S) :- holds(-P v -Q,S).
holds(-(P v Q),S) :- holds(-P & -Q,S).
holds(-(P => Q),S) :- holds(-(-P v Q),S).
holds(-(P <=> Q),S) :- holds(-((P => Q) & (Q => P)),S).
holds(-all(V,P),S) :- holds(some(V,-P),S).
holds(-some(V,P),S) :- not holds(some(V,P),S). /* Negation */
holds(-P,S) :- isatom(P), not holds(P,S).      /* by failure */
holds(all(V,P),S) :- holds(-some(V,-P),S).
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).


/* The following clause treats the holds predicate for all atoms,
   including Prolog system predicates. For this to work properly,
   the GOLOG programmer must provide, for all atoms taking a
   situation argument, a clause giving the result of restoring
   its suppressed situation argument, for example:
   restoreSitArg(ontable(X),S,ontable(X,S)).
*/
holds(true,_).
holds(A,S) :-
   restoreSitArg(A,S,F), F
   ;
   not restoreSitArg(A,S,F), isatom(A), A.

isatom(A) :- not (A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
    A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W)).


/* holdsBelief/2

   Defining when a proposition holds to a degree
   (see holdsBelief/3 next)
*/
holdsBelief(-(true),_) :- !, false.
holdsBelief(true,_).
holdsBelief(Q,[(S,_)]) :-
   holds(Q,S).
holdsBelief(Q,[Head | Tail]) :-
   Head = (S,_),
   holds(Q,S),!
   ;
   holdsBelief(Q,Tail).
holdsBelief(-Q,B) :- not holdsBelief(Q,B).
```

```
/* holdsBelief/3

   holdsBelief(Sentence,Degree,BeliefState), takes as input a Sentence
   and the agent's current belief state, and returns the Degree to
   which the agent believes the sentence.
*/
holdsBelief(P,Degree,B) :- holdsBelief(P,B),!, belief(P,Degree,B).
holdsBelief(-P,Degree,B) :- holdsBelief(-P,S),!, belief(-P,Degree,B).




/* ---- BDI Controller (BDI-POP) and Simulation environ. (FireEater world): ---- */


/* Call

   startSimulation(SimTime, RateOfChange, AgentSpeed, Points):
   Inputs:
      SimTime: minimum simulation time in seconds (e.g., 20)
      RateOfChange: # obstacle changes per cycle (e.g., 3)
      AgentSpeed: # agent moves allowed per cycle (e.g., 2)
   Outputs:
      Points
*/
startSimulation(SimTime, RateOfChange, AgentSpeed, Points) :-
   worldInfo(WorldInfo),
   WorldInfo = (NuofObstacles, NuofFires, NuofPowPacs,   WorldDimension),
   initializeWorld(NuofObstacles, NuofFires, NuofPowPacs, WorldDimension,
                                                          InitAgentPos),
   makeAssertions(InitAgentPos),
   agentInfo(AgentInfo),
   AgentInfo = (Beh, Progs, Init_Beliefs, Hor),
   NewAgentInfo = (Beh, Progs, Hor),
   get_flag(unix_time, StartTime),
   doloop_simulation(WorldDimension, NewAgentInfo, Init_Beliefs, [],
                     SimTime, StartTime,
                     RateOfChange, AgentSpeed, Points, 0,
                     intn(_,nil,_,_), stop, go).

doloop_simulation(WD, AI, Bels, Ints, SimTime, StartTime,
                  ROC, AS, Points, Points_SF, ActiveInt, Pol, Stop_Go) :-
   (get_flag(unix_time, CurrentTime),
   TimeElapsed is CurrentTime - StartTime,
   TimeElapsed >= SimTime
   ;
   Stop_Go == stop), !,
```

```
      Points = Points_SF
      ;
      doloop_evolve(WD, AI, Bels, Ints, SimTime, StartTime,
                    ROC, 0, AS, Points, Points_SF, ActiveInt, Pol).

doloop_evolve(WD, AI, Bels, Ints, SimTime, StartTime,
                ROC, ChangesSoFar, AS, Points, Points_SF, ActiveInt, Pol) :-
   ChangesSoFar =:= ROC, !,
      doloop_agent(WD, AI, Bels, Ints, SimTime, StartTime,
                   ROC, AS, 0, Points, Points_SF, ActiveInt, Pol)
   ;
   NewChangesSoFar is ChangesSoFar + 1,
   write("evv: "), writeln(NewChangesSoFar),
   evolveSimWorld(WD),
   doloop_evolve(WD, AI, Bels, Ints, SimTime, StartTime,
                 ROC, NewChangesSoFar, AS, F_FL, FL_SF, ActiveInt, Pol).

doloop_agent(WD, AI, Bels, Ints, SimTime, StartTime,
               ROC, AS, AS_SF, Points, Points_SF, ActiveInt, Pol) :-
   (
   get_flag(unix_time, CurrentTime),
   TimeElapsed is CurrentTime - StartTime,
   TimeElapsed >= SimTime
   ;
   AS_SF =:= AS
   ), !,
      doloop_simulation(WD, AI, Bels, Ints, SimTime, StartTime,
                        ROC, AS, Points, Points_SF, ActiveInt, Pol, go)
      ;
      AI = (_, _, Hor),
      ActiveInt = intn(Name, Prog, Achieves, Value),
      (Pol == stop, !, % reached end of policy
         (Prog == nil, !, % if prog is finished, simply get next intention:
            deliberate(WD, AI, Bels, Ints, SimTime, StartTime,
                       ROC, AS, AS_SF, Points, Points_SF)
            ;
            getRestProg(Prog, Hor, RestProg),
            bestDoPo(Prog, Name, Bels, Hor, Pol2, _, _), !,
            NewActiveInt = intn(Name, RestProg, Achieves, Value),
            (Pol2 == stop, !, % the planner could not gen. a usable policy
               deliberate(WD, AI, Bels, Ints, SimTime, StartTime,
                          ROC, AS, AS_SF, Points, Points_SF)
               ;
               getFirstAction(Pol2, Action, Rest_Pol),
               getNaturesChoice(Action, NaturesChoice),
               executeInSimWorld(NaturesChoice, Points_SF, NewPoints_SF),
               NewAS_SF is AS_SF + 1,
```

```
      write("agt: "), writeln(NewAS_SF),
      % acquire the observation from simworld, given the action outcome
      getPercept(NaturesChoice, SensedValue),
      perceive(Action, SensedValue, Observation),
      getSubPolicy(Rest_Pol, Observation, NewPol),
      beliefUpdate(Observation, Action, NewBels, Bels),
      % beliefUpdate is wrt. prev. fires list, so bel. up.
      % must be done before updating Fires list
      (NaturesChoice == eat,
         simPossAct(eat), !,
         initAgentPos(Pos),
         agAtt(Pos,InitSit),
         FiresList(NewFiresList),
         retract(futureFiresList(_,_)),
         assert(futureFiresList(NewFiresList, InitSit))
         ;
      NaturesChoice == grab,
         simPossAct(grab), !,
         initAgentPos(Pos),
         agAtt(Pos,InitSit),
         powpacList(NewPowPacList),
         retract(futurePowPacList(_,_)),
         assert(futurePowPacList(NewPowPacList, InitSit))
         ;
      true
      ),
      (NewBels == [], !,
         writeln("NewBels == []"),
         doloop_simulation(_, _, _, _, SimTime, SimTime,
                           _, _, Points, NewPoints_SF, _, _, stop)
         ;
         doloop_agent(WD, AI, NewBels, Ints, SimTime, StartTime,
                      ROC, AS, NewAS_SF,
                      Points, NewPoints_SF, NewActiveInt, NewPol)
      )% end NewBels == ...
   )% end Pol2 == ...
)% end Prog == ...

; % Pol =\= stop

getFirstAction(Pol, Action, Rest_Pol),
getNaturesChoice(Action, NaturesChoice),
executeInSimWorld(NaturesChoice, Points_SF, NewPoints_SF),
NewAS_SF is AS_SF + 1,
write("agt: "), writeln(NewAS_SF),
% acquire the observation from simworld, given the action outcome
getPercept(NaturesChoice, SensedValue),
```

```
        perceive(Action, SensedValue, Observation),
        getSubPolicy(Rest_Pol, Observation, NewPol),
        beliefUpdate(Observation, Action, NewBels, Bels),
        (NaturesChoice == eat,
             simPossAct(eat), !,
             initAgentPos(Pos),
             agAtt(Pos,InitSit),
             FiresList(NewFiresList),
             retract(futureFiresList(_,_)),
             assert(futureFiresList(NewFiresList, InitSit))
             ;
           NaturesChoice == grab,
              simPossAct(grab), !,
              initAgentPos(Pos),
              agAtt(Pos,InitSit),
              powpacList(NewPowPacList),
              retract(futurePowPacList(_,_)),
              assert(futurePowPacList(NewPowPacList, InitSit))
              ;
           true
           ),
           (NewBels == [], !,
              writeln("NewBels == []"),
              doloop_simulation(_, _, _, _, SimTime, SimTime,
                                _, _, Points, NewPoints_SF, _, _, stop)
              ;
              doloop_agent(WD, AI, NewBels, Ints, SimTime, StartTime,
                           ROC, AS, NewAS_SF,
                           Points, NewPoints_SF, ActiveInt, NewPol)
           )% end NewBels == ...
        ).% end Pol == ...

deliberate(WD, AI, Bels, Ints, SimTime, StartTime, ROC, AS, AS_SF, F_FL, FL_SF) :-
   AI = (Behs, Dess, Hor),
   (Ints == [], !,
      focus(Bels, Behs, Dess, NewInts, Hor) % output: NewInts
      ;
      NewInts = Ints
   ),
   popIntentionStack(ActiveInt, NewInts, PoppedInts),
   doloop_agent(WD, AI, Bels, PoppedInts, SimTime, StartTime,
                ROC, AS, AS_SF, F_FL, FL_SF, ActiveInt, stop).
```

```
%---- Definitions of some important clauses ----%

initializeWorld(NuofObstacles, NuofFires, NuofPowPacs, WorldDimension, AgentPos) :-
   placeObstacles(NuofObstacles, 0, [], WorldDimension),
   placeFires(NuofFires, 0, [], WorldDimension),
   placePowPacs(NuofPowPacs, 0, [], WorldDimension),
   placeAgent(WorldDimension, AgentPos).

makeAssertions(InitAgentPos) :-
   (retract(initAgentPos(_)) ; true),
   assert(initAgentPos(InitAgentPos)),
   FiresList(FiresList),
   powpacList(PowPacList),
   (retract_all(futureFiresList(_,_)) ; true),
   (retract_all(futurePowPacList(_,_)) ; true),
   agAtt(InitAgentPos, S),
   assert(futureFiresList(FiresList, S)),
   assert(
   (futureFiresList(NewFiresList, doo(A, Ss)) :-
       futureFiresList(OldFiresList, Ss),
       (A = eat, !,
          (possAct(eat, Ss), !,
             agAtt(loc(X,Y), Ss),
             removeFromList((X,Y),OldFiresList,NewFiresList)
             ;
             NewFiresList = OldFiresList
          )
          ;
          NewFiresList = OldFiresList
      )
   )
   ),
   assert(futurePowPacList(PowPacList, S)),
   assert(
   (futurePowPacList(NewPowPacList, doo(A, Ss)) :-
       futurePowPacList(OldPowPacList, Ss),
       (A = grab, !,
          (possAct(grab, Ss), !,
             agAtt(loc(X,Y), Ss),
             removeFromList((X,Y),OldPowPacList,NewPowPacList)
             ;
             NewPowPacList = OldPowPacList
          )
          ;
          NewPowPacList = OldPowPacList
       )
   )
```

```
   ).

focus(Bels, Behs, Dess, Ints3, Hor) :-
   % get best progs
   focusAux(Bels, Behs, Dess, [], Ints2, Hor),
   % order progs
   sort(4,>=,Ints2,Ints3).

focusAux(_, [], _, IntsSoFar, IntsSoFar, _).
focusAux(Bels, Behs, Dess, IntsSoFar, Ints3, Hor) :-
   Behs = [Beh|Rest],
   % find all desires that achieve same behavior
   findall(X,(member(X,Dess),X=intn(_,_,achieves(Beh),_)),Bag),
   findMostValuedProg(Bels,Bag,MostValuedProg,Hor),
   NewIntsSoFar = [MostValuedProg|IntsSoFar],
   focusAux(Bels, Rest, Dess, NewIntsSoFar, Ints3, Hor).

findMostValuedProg(Bels,CandidateInts,BestInt,Hor) :-
   CandidateInts = [First|Rest],
   mostValuedAux(Bels,Rest,First,intn(_,_,_,-1000),Hor,BestInt).

mostValuedAux(Bels,[],Last,BestIntSoFar,Hor,BestInt) :-
   Last = intn(Name,Prog,_,_),
   ShallowHor is Hor - 1,
   bestDoPo(Prog,Name,Bels,ShallowHor,_,LastValue,_),
   Last = intn(_,_,_,LastValue),
   BestIntSoFar = intn(_,_,_,BestValueSoFar),
   (LastValue > BestValueSoFar, !,
      BestInt = Last
      ;
      BestInt = BestIntSoFar
   ).
mostValuedAux(Bels,CandidateInts,NextInt,BestIntSoFar,Hor,BestInt) :-
   NextInt = intn(Name,Prog,_,_),
   ShallowHor is Hor - 1,!,
      bestDoPo(Prog,Name,Bels,ShallowHor,_,NextValue,_),
      NextInt = intn(_,_,_,NextValue),
      BestIntSoFar = intn(_,_,_,BestValueSoFar),
      (NextValue > BestValueSoFar, !,
         NewBestIntSoFar = NextInt
         ;
         NewBestIntSoFar = BestIntSoFar
      ),
      CandidateInts = [NewNext|Rest],
      mostValuedAux(Bels,Rest,NewNext,NewBestIntSoFar,Hor,BestInt).
```

```
placeObstacles(NuofObstacles, NOO_SF, ObstacleList_SF, WorldDimension) :-
NOO_SF == NuofObstacles, !,
(retract_all(obstacleList(_)) ; true),
assert(obstacleList(ObstacleList_SF))
%write("Obstacles: "), writeln(ObstacleList_SF)
;
random(N1), random(N2),
mod(N1,WorldDimension,M1), mod(N2,WorldDimension,M2),
Mm1 is M1 + 1, Mm2 is M2 + 1,
(member((Mm1,Mm2),ObstacleList_SF), !,
placeObstacles(NuofObstacles, NOO_SF,
                               ObstacleList_SF, WorldDimension)
;
NewNOO_SF is NOO_SF + 1,
NewObstacleList_SF = [(Mm1,Mm2)|ObstacleList_SF],
placeObstacles(NuofObstacles, NewNOO_SF,
                               NewObstacleList_SF, WorldDimension)
).

placeFires(NuofFires, NOF_SF, FiresList_SF, WorldDimension) :-
   NOF_SF == NuofFires, !,
      (retract_all(FiresList(_)) ; true),
      assert(FiresList(FiresList_SF))
      ;
      random(N1), random(N2),
      mod(N1,WorldDimension,M1), mod(N2,WorldDimension,M2),
      Mm1 is M1 + 1, Mm2 is M2 + 1,
      obstacleList(ObstacleList),
      ((member((Mm1,Mm2),ObstacleList) ; member((Mm1,Mm2),FiresList_SF)), !,
         placeFires(NuofFires, NOF_SF, FiresList_SF, WorldDimension)
         ;
         NewNOF_SF is NOF_SF + 1,
         NewFiresList_SF = [(Mm1,Mm2)|FiresList_SF],
         placeFires(NuofFires, NewNOF_SF, NewFiresList_SF, WorldDimension)
      ).

placePowPacs(NuofPowPacs, NOP_SF, PowPacList_SF, WorldDimension) :-
   NOP_SF == NuofPowPacs, !,
      (retract_all(powpacList(_)) ; true),
      assert(powpacList(PowPacList_SF))
      ;
      random(N1), random(N2),
      mod(N1,WorldDimension,M1), mod(N2,WorldDimension,M2),
      Mm1 is M1 + 1, Mm2 is M2 + 1,
      obstacleList(ObstacleList), FiresList(FiresList),
      ((member((Mm1,Mm2),ObstacleList)
        ; member((Mm1,Mm2),PowPacList_SF)
```

```
        ; member((Mm1,Mm2),FiresList)), !,
           placePowPacs(NuofPowPacs, NOP_SF, PowPacList_SF, WorldDimension)
           ;
           NewNOP_SF is NOP_SF + 1,
           NewPowPacList_SF = [(Mm1,Mm2)|PowPacList_SF],
           placePowPacs(NuofPowPacs, NewNOP_SF, NewPowPacList_SF, WorldDimension)
      ).

placeAgent(WorldDimension, AgentPos) :-
   random(N1), random(N2),
   mod(N1,WorldDimension,M1), mod(N2,WorldDimension,M2),
   Mm1 is M1 + 1, Mm2 is M2 + 1,
   obstacleList(ObstacleList), powpacList(PowPacList), FiresList(FiresList),
   ((member((Mm1,Mm2),ObstacleList)
    ; member((Mm1,Mm2),PowPacList)
    ; member((Mm1,Mm2),FiresList)), !,
      placeAgent(WorldDimension, AgentPos)
      ;
      AgentPos = loc(Mm1,Mm2),
      (retract(simAtt(_)) ; true),
      assert(simAtt(AgentPos))
   ).

evolveSimWorld(WorldDimension) :-
   WD = WorldDimension,
   obstacleList(OL),
   FiresList(FL),
   powpacList(PL),
   shiftObstacle(OL, FL, PL, NewObstacleList, WD),
   retract(obstacleList(_)),
   assert(obstacleList(NewObstacleList)).

shiftObstacle(ObstacleLt, FireLt, PowPacLt, NewObstacleLt, WorldDimension) :-
   random(N1), random(N2),
   mod(N1,WorldDimension,M1), mod(N2,WorldDimension,M2),
   Mm1 is M1 + 1, Mm2 is M2 + 1,
   member((Mm1,Mm2),ObstacleLt), !,
      removeFromList((Mm1,Mm2),ObstacleLt,NewObstacleLt1),
      placeOneObject(NewObstacleLt1,FireLt,PowPacLt,
                     NewObstacleLt,WorldDimension)
      ;
      shiftObstacle(ObstacleLt, FireLt, PowPacLt,
                    NewObstacleLt, WorldDimension).

removeFromList((X,Y),List,NewList) :-
   bagof((Xx,Yy),(member((Xx,Yy),List), (Xx,Yy) \= (X,Y)),NewList).
```

```
placeOneObject(ObjectLt1, ObjectLt2, ObjectLt3, NewObjectLt, WorldDimension) :-
   random(N1), random(N2),
   mod(N1,WorldDimension,M1), mod(N2,WorldDimension,M2),
   Mm1 is M1 + 1, Mm2 is M2 + 1,
   (
      not member((Mm1,Mm2),ObjectLt1),
      not member((Mm1,Mm2),ObjectLt2),
      not member((Mm1,Mm2),ObjectLt3),
      not simAtt(loc(Mm1,Mm2)), !,
      NewObjectLt = [(Mm1,Mm2)|ObjectLt1]
      ;
      placeOneObject(ObjectLt1, ObjectLt2, ObjectLt3, NewObjectLt, WorldDimension)
   ).

% This procedure assumes that all while conditions are 'true' and not contingent.
% Thus, all BDI-POP((R)) architectures only allow
  input programs with 'true' while conditions.
getRestProg(Prog, Hor, RestProg) :- getRestProgAux(Prog, Hor, 1, RestProg).

getRestProgAux(Prog, Hor, Depth, RestProg) :-
   Prog = (while(true,A) : B), !,
      getRestProgAux(A:while(true,A):B, Hor, Depth, RestProg)
      ;
      (
      Prog = (E1 : E2), E1 \= (E11 : E12), !,
         (Depth == Hor, !,
            RestProg = E2
            ;
            NewDepth is Depth + 1,
            getRestProgAux(E2, Hor, NewDepth, RestProg)
         )
         ;
         Prog = (E1 : E2), !,
            E1 = (E11 : E12),
            getRestProgAux(E11:(E12:E2), Hor, Depth, RestProg)
            ;
            RestProg = nil
      ).

getSubPolicy(stop, _, stop).
getSubPolicy(Pol, Observation, SubPol) :-
(Pol = ( ?(observationIs(Observation),SubPol) ), !
)
;
(Pol = ( if( observationIs(Observation_PH), SubPol1, SubPol2) ),
(
Observation_PH == Observation, !,
```

```
SubPol = SubPol1
;
getSubPolicy(SubPol2, Observation, SubPol)
)
).

getFirstAction(Pol, Action, Rest_Pol) :-
   ...

getNaturesChoice(Action, NaturesChoice) :-
   ...

executeInSimWorld(NaturesChoice, FirePoints, NewFirePoints) :-
   ...

simPossAct(...) :-
   ...

simProbNat(...) :-
   ...

simProbObs(Observation,Action,Probability) :-
   ...

getPercept(Action, SensedValue) :-
   ...

perceive(Action, SensedValue, Observation) :-
   ...




/* ---- Problem description ---- */

/* FireEater world: a 10 by 10 grid world, with obstacles, fire and power packs.


/* Successor-state axiom;
   agAtt(loc(X,Y), S) denotes whether the agent is at loc(X,Y) in situation S)
   senseloc and eat are implicitly handled at the last line.
*/
agAtt(loc(1,1),s1).
...
agAtt(loc(10,10),s100).
```

```
agAtt(loc(X,Y), doo(A, S)) :-
   possAct(A, S), !,
      (
         A == left, !, agAtt(loc(Xp1,Y), S), X is Xp1 - 1
         ;
         A == right, !, agAtt(loc(Xm1,Y), S), X is Xm1 + 1
         ;
         A == up, !, agAtt(loc(X,Ym1), S), Y is Ym1 + 1
         ;
         A == down, !, agAtt(loc(X,Yp1), S), Y is Yp1 - 1
         ;
         agAtt(loc(X,Y), S)
      )
      ;
      agAtt(loc(X,Y), S).


/* ---- Initial Database ---- */

/* Initial world state

   The real initial position of the agent is determined randomly.
   The agent initially believes it is at (only) the real initial position.
*/
% See startSimulation().


/* Initial belief state:
   The probabiity that the agent is in some situation,
   for each of the initial situations
*/
% See startSimulation().


/* Agent actions available:
*/
agentAction(up).
agentAction(down).
agentAction(left).
agentAction(right).
agentAction(noop).
agentAction(eat).
agentAction(grab).
agentAction(senseloc).
```

```
/* Agent desires:
   Primitive motivations; the things the agent wants to achieve
*/
desires(Desires) :- Desires = [findFire, eating, findPowpac, grabing].


/* Agent behaviors:
   Various programs that the agent can choose from to achieve behaviors
*/
behaviors(Behaviors) :- Behaviors = [
   intn(findFire1, (senseloc :
                    (senseloc # left # right # up # down # noop) :
                    (senseloc # left # right # up # down # noop) :
                    nil
                   ),
                   achieves(findFire), _),
   intn(findFire2, ((left # right # up # down # noop) :
                    (left # right # up # down # noop) :
                    (left # right # up # down # noop) :
                    nil
                   ),
                   achieves(findFire), _),
   intn(eat1, ((eat # left # right # up # down # noop) :
               (eat # left # right # up # down # noop) :
               (eat # left # right # up # down # noop) :
                nil
              ),
              achieves(eating), _),
   intn(findPowpac1, (senseloc :
                      (senseloc # left # right # up # down # noop) :
                      (senseloc # left # right # up # down # noop) :
                      nil
                     ),
                      achieves(findPowpac), _),
   intn(findPowpac2, ((left # right # up # down # noop) :
                      (left # right # up # down # noop) :
                      (left # right # up # down # noop) :
                      nil
                     ),
                      achieves(findPowpac), _),
   intn(grab1, ((grab # left # right # up # down # noop) :
                (grab # left # right # up # down # noop) :
                (grab # left # right # up # down # noop) :
                nil
               ),
               achieves(grabing), _)
   ].
```

```
/* Reward functions:
   Each program has an associated reward function
*/
% To find fires, the agent moves towards areas where it is surrounded by fires.
reward(findFire1,R,S) :-
   agAtt(loc(X,Y),S),
   futureFiresList(FiresList, S),
   (Xx is X-1, member((Xx,Y),FiresList), !,
      R1 = 1
      ;
      R1 = 0
   ),
   (Xx is X+1, member((Xx,Y),FiresList), !,
      R2 = 1
      ;
      R2 = 0
   ),
  (Yy is Y-1, member((X,Yy),FiresList), !,
      R4 = 1
      ;
      R4 = 0
   ),
   (Yy is Y+1, member((X,Yy),FiresList), !,
      R5 = 1
      ;
      R5 = 0
   ),
   (Xx is X-1, Yy is Y-1, member((Xx,Yy),FiresList), !,
      R6 = 1
      ;
      R6 = 0
   ),
   (Xx is X-1, Yy is Y+1, member((Xx,Yy),FiresList), !,
      R7 = 1
      ;
      R7 = 0
   ),
   (Xx is X+1, Yy is Y-1, member((Xx,Yy),FiresList), !,
      R8 = 1
      ;
      R8 = 0
   ),
   (Xx is X+1, Yy is Y+1, member((Xx,Yy),FiresList), !,
      R9 = 1
      ;
```

```
      R9 = 0
   ),
   R is R1 + R2 + R4 + R5 + R6 + R7 + R8 + R9.
reward(findFire2,R,S) :-
...
reward(eat1,R,S) :-
(S = doo(A,_), A = eat, !,
R = 100
;
R = 0
).
% To find power pacs, the agent moves towards areas
  where it is surrounded by power pacs.
reward(findPowpac1,R,S) :-
   ...
reward(findPowpac2,R,S) :-
   ...
reward(grab1,R,S) :-
   (S = doo(A,_), A = grab, !,
      R = 103
      ;
      R = 0
   ).


/* Stochastic actions have a finite number of outcomes; we list all of them:
   choiceNat(N,A,S)
   Changes here requires changes to probNat(.) and to getNaturesChoice(.).
*/
choiceNat(left,left,_).
choiceNat(up,left,_).
choiceNat(down,left,_).

choiceNat(right,right,_).
choiceNat(up,right,_).
choiceNat(down,right,_).

choiceNat(up,up,_).
choiceNat(right,up,_).
choiceNat(left,up,_).

choiceNat(down,down,_).
choiceNat(left,down,_).
choiceNat(right,down,_).

choiceNat(eat,eat,_).
choiceNat(grab,grab,_).
```

```
choiceNat(noop,noop,_).
choiceNat(senseloc,senseloc,_).


/* Each action is associated with some (one or more; finite) observations;
   we list all of them:
   choiceObs(A, ListOfObservations)
*/
choiceObs(left, [obsnil] ).
choiceObs(right, [obsnil] ).
choiceObs(up, [obsnil] ).
choiceObs(down, [obsnil] ).
choiceObs(eat, [obsnil] ).
choiceObs(grab, [obsnil] ).
choiceObs(noop, [obsnil] ).


% For 10 by 10 grid world
choiceObs(senseloc, ObsList) :-
   List = [1,2,3,4,5,6,7,8,9,10],
   findall(obsloc(X,Y),(member(X,List), member(Y,List)),ObsList).


/* The state transition function is defined by several predicates;
   (probNat(N,A,P,S):
   the probability P that nature will choose action N, given action A in S)
*/
probNat(left,left,0.95,_).
probNat(up,left,0.025,_).
probNat(down,left,0.025,_).

probNat(right,right,0.95,_).
probNat(up,right,0.025,_).
probNat(down,right,0.025,_).

probNat(up,up,0.95,_).
probNat(right,up,0.025,_).
probNat(left,up,0.025,_).

probNat(down,down,0.95,_).
probNat(left,down,0.025,_).
probNat(right,down,0.025,_).

probNat(eat,eat,1,_).
probNat(grab,grab,1,_).
probNat(noop,noop,1,_).
probNat(senseloc,senseloc,1,_).
```

```
/* The observation function is defined by several predicates;
   (probObs(O,A,P,S):
   the probability P that the agent will observe O in S, given action A)
*/
probObs(obsnil,A,1.0,_) :-
  (A == left ; A == right ; A == up ; A == down ; A == eat ; A == grab ; A == noop).
probObs(obsloc(X,Y),senseloc,P,S) :-
   agAtt(loc(Xx,Yy),S),
   (
   X =:= Xx,Y =:= Yy,P is 0.96;
   X =:= Xx-1,   Y =:= Yy-1,  P is 0.005;
   X =:= Xx-1,   Y =:= Yy+1,  P is 0.005;
   X =:= Xx-1,   Y =:= Yy,    P is 0.005;
   X =:= Xx+1,   Y =:= Yy-1,  P is 0.005;
   X =:= Xx+1,   Y =:= Yy+1,  P is 0.005;
   X =:= Xx+1,   Y =:= Yy,    P is 0.005;
   X =:= Xx,     Y =:= Yy-1,  P is 0.005;
   X =:= Xx,     Y =:= Yy+1,  P is 0.005
   ), !
   ;
   P is 0.


/* We formulate precondition axioms using the predicate
   possAct(Action, Situation) for actions, and
   possObs(Observation, Sense-action, Situation) for observations.
   The right-hand side of precondition axioms provides conditions
   under which Action or Observation is possible in Situation
*/
possAct(left,S) :-
   agAtt(loc(X,Y),S), !,
   (X =\= 1 ; Xx is X-1, obstacleList(ObstacleList), !,
   not member((Xx,Y),ObstacleList)).
possAct(right,S) :-
   agAtt(loc(X,Y),S), !,
   (X =\= 10 ; Xx is X+1, obstacleList(ObstacleList), !,
   not member((Xx,Y),ObstacleList)).
possAct(up,S) :-
   agAtt(loc(X,Y),S), !,
   (Y =\= 10 ; Yy is Y+1, obstacleList(ObstacleList), !,
   not member((X,Yy),ObstacleList)).
possAct(down,S) :-
   agAtt(loc(X,Y),S), !,
   (Y =\= 1 ; Yy is Y-1, obstacleList(ObstacleList), !,
   not member((X,Yy),ObstacleList)).
```

```
possAct(eat,S) :-
   agAtt(loc(X,Y),S), !,
   futureFiresList(FiresList, S), !, member((X,Y),FiresList).
possAct(grab,S) :-
   agAtt(loc(X,Y),S), !,
   futurePowPacList(PowPacList, S), !, member((X,Y),PowPacList).
possAct(noop,_).
possAct(senseloc,_).

possObs(obsnil,A,_) :-
   A == left ; A == right ; A == up ; A == down ; A == eat ; A == grab ; A == noop.
possObs(obsloc(X,Y),senseloc,S) :-
   agAtt(loc(Xx,Yy),S), !,
   X1 is X-Xx, abs(X1,XAbs), XAbs =< 1,
   Y1 is Y-Yy, abs(Y1,YAbs), YAbs =< 1,
   X >= 1, X =< 10, Y >= 1, Y =< 10,
   obstacleList(ObstacleList), !, not member((X,Y),ObstacleList).


%---- Some world parameters ----%

worldInfo(WorldInfo) :-
   NuofObstacles = 24,
   NuofFires = 18,
   NuofPowPacs = 18,
   36 is NuofFires + NuofPowPacs,
   WorldDimension = 10,
   WorldInfo = (NuofObstacles, NuofFires, NuofPowPacs, WorldDimension).


%---- Some agent parameters ----%

agentInfo(AgentInfo) :-
   behaviors(Behaviors),
   desires(Desires),
   initAgentPos(Pos),
   agAtt(Pos,S),
   Init_Beliefs=[(S,1.0)],
   Hor = 3,
   AgentInfo = (Desires, Behaviors, Init_Beliefs, Hor).
```

Appendix C

Paper 1 – Extending DTGolog to Deal with POMDPs

Conference paper [85] resulting from work in this dissertation.

G. Rens, A. Ferrein, and E. van der Poel. Extending DTGolog to deal with POMDPs. In F. Nicolls, editor, *Proc. of 19th Annual Symposium of the Pattern Recognition Association of South Africa (PRASA-08)*, pages 49–54, Cape Town, South Africa, 2008. UCT Press. url: http://hdl.handle.net/10204/2972.

The paper starts on the next page.

Extending DTGolog to Deal with POMDPs

Gavin Rens[1,2], Alexander Ferrein[3], Etienne van der Poel[1]

[1] School of Computing, Unisa, Pretoria, South Africa
[2] Knowledge Systems Group, Meraka Institute, CSIR, Pretoria, South Africa
[3] Knowledge-Based Systems Group, RWTH Aachen University, Aachen, Germany

grens@csir.co.za
ferrein@cs.rwth-aachen.de
evdpoel@unisa.ac.za

Abstract

For sophisticated robots, it may be best to accept and reason with noisy sensor data, instead of assuming complete observation and then dealing with the effects of making the assumption. We shall model uncertainties with a formalism called the *partially observable Markov decision process* (POMDP). The planner developed in this paper will be implemented in *Golog*; a theoretically and practically 'proven' agent programming language. There exists a working implementation of our POMDP-planner.

1. Introduction

If a robot or agent can perceive every necessary detail of its environment, its model is said to be *fully observable*. In many practical application this assumption is good enough for the agent to fulfill its tasks; it is nevertheless unrealistic. A more accurate model is a *partially observable model*. The agent takes into account that its sensors are imperfect, and that it does not know every detail of the world. That is, the agent can incorporate the probabilities of errors associated with its sensors, and other uncertainties inherent in perception in the real world, for example, obscured objects. If an agent or robot cannot represent the uncertainties inherent in perception, it has to *assume* perfect perception; this assumption either might leads to spurious conclusions or the necessity for additional methods that keep the agent's reasoning reasonable. For sophisticated robots or agents, it may be best to accept and reason with noisy sensor data.

One model for reasoning under uncertainty with partial observability is the *partially observable Markov decision process* (POMDP). In this paper we present POMDP models based on the robot programming and planning language Golog [1]. In particular, we extend DTGolog [2], a Golog dialect. DTGolog employs a notion of perfect perception; we extent it with a notion of graded belief.

The rest of the paper is organised as follow. In the next section we briefly introduce the situation calculus and present the robot programming and planning language DTGolog, before we formally define POMDPs in Section 3. In Section 4 we present some related work. Section 5 introduces the predicate $BestDoPO$ which defines the semantics of the POMDP planner in Golog. Section 6 presents an simple example of how planning under partial observability is conducted. We conclude with Section 7.

2. The Situation Calculus and DTGolog

The situation calculus is a first order logic dialect for reasoning about dynamical systems based on agent actions. The outcomes of a bout of reasoning in the situation calculus are meant to have effects on the environment outside the agent. When an agent or robot performs an action, the truth value of certain predicates may change. Predicates whose value can change due to actions are called *fluents*. Fluents have the *situation term* s as the last argument.

A special function symbol do is defined in the situation calculus. $do(a, s)$ is the name of the situation (that the agent is in) given the agent does action a in situation s. Note that $do(a_2, do(a_1, s))$ is also a situation term, where a_2 and a_1 are actions.

To reason in the situation calculus, one needs to define an initial knowledge base (KB). The only situation term allowed in the initial KB is the special *initial situation* S_0. S_0 is the situation before any action has been done.

There are two more formulas that need our attention:

1. The *precondition axioms* are formulas of the form $Poss(a, s)$, which means action a is possible in situation s ($\neg Poss(a, s)$ means it is not possible). Precondition axioms need to be defined for each action.
2. *Successor-state axioms* are formulas that define how fluents' values change due to actions. There needs to be a successor-state axiom for each fluent, and each such successor-state axiom mentions only the actions that have an effect on the particular fluent.

Please refer to [3] for a detailed explication of the situation calculus, including a description of the famous *frame problem* and how the *basic action theory* is a solution to this problem. Alternatively, refer to [4] for a one-chapter coverage of the situation calculus.

Decision-theoretic Golog (DTGolog) [2] is an extension to Golog to reason with probabilistic models of uncertain actions. The formal underlying model is that of fully observable Markov decision processes (MDPs).

Golog is an agent programming language (APL) developed by [1]. It is based on the situation calculus. It has most of the constructs of regular procedural programming languages (iteration, conditionals, etc.). What makes it different from other programming languages is that it is used to specify and control *actions* that are intended to be executed in the real world or a simulation of the real world. That is, Golog's main variable type is the action (not the number).

Complex actions can be specified by combining atomic actions. The following are all complex actions (where a subscripted is an atomic action and φ is a sentence):

- while φ do a_1 (iteration of actions);
- ? $\varphi : a_1$ (test action);
- if φ then a_1 else a_2 (conditional actions);
- $a_1; a_2; \ldots; a_k$ (sequence of actions);
- $a_1 \mid a_2$ (nondeterministic choice of actions);
- $\wp x.(a_1)$ (nondeterministic finite choice of arguments—of x in a_1);

$Do(A, s, s')$ holds if and only if the complex action A can terminate legally in s' when started in situation s.

The DTGolog algorithm is defined with $BestDo$ predicates, taking on the role of Golog's Do. The DTGolog interpreter however, does not simply 'perform' the program (complex action) given it, but calculates an optimal policy based on an optimization theory: the forward search value iteration algorithm for fully observable MDPs. [1] capture the nondeterministic aspect of MDPs with predicates $stochastic$, and $prob$. $prob(n, p, s)$ determines the probability p with which n is the outcome in some situation s. (In this section we define prob as a *function* that *returns* the probability.) Let $choice'(a) \doteq \{n_1, \ldots, n_k\}$ (derived from $stochastic$) be the k actions that nature could 'choose' (the actual action performed) for the agent's *intended* action a. For stochastic action a,

$$BestDo(a; rest, s, h, \pi, v, pr) \doteq$$
$$\exists \pi', v'. BestDoAux(choice'(a), a, rest, s, h, \pi', v', pr) \wedge$$
$$\pi = a; senseEffect(a), \pi' \wedge v = reward(s) + v'.$$

$a; r$ is the input program, with a the first action in the program and r the rest of the program; s is the situation term; the agent designer needs to set the number of steps (actions) h for which a policy is sought—the *planning horizon*; π returns the policy; v is the expected reward for executing π; pr returns the probability with which the input program will be executed as specified, given the policy and given the effects of the environment. $senseEffect(a)$ is a pseudo-action included in the formalism to ensure that the formalism stays in the fully observable MDP model. $BestDoAux$ deals with each of the possible realizations of a stochastic action:

$$BestDoAux(\{n_1, \ldots, n_k\}, a, r, s, h, \pi, v, pr) \doteq$$
$$\neg Poss(n_1, s) \wedge BestDoAux(\{n_2, \ldots, n_k\},$$
$$a, r, s, h, \pi, v, pr) \vee Poss(n_1, s) \wedge$$
$$\exists \pi', v', pr'. BestDoAux(\{n_2, \ldots, n_k\}, a, r, s, h, \pi', v', pr') \wedge$$
$$\exists \pi_1, v_1, pr_1. BestDo(r, do(n_1, s), h - 1, \pi_1, v_1, pr_1) \wedge$$
$$senseCond(n_1, \varphi_1) \wedge \pi = \textbf{if } \varphi_1 \textbf{ then } \pi_1 \textbf{ else } \pi' \textbf{ endif} \wedge$$
$$v = v' + v_1 \cdot prob(n_1, a, s) \wedge pr = pr' + pr_1 \cdot prob(n_1, a, s).$$

For any action n, $senseCond(n, \varphi)$ supplies a sentence φ that is placed in the policy being generated. φ holds if and only if the value returned by the sensor can verify that action n was performed.

When either of two actions δ_1 and δ_2 can be performed, the policy associated with the action that produces the greater value (current sum of rewards) is preferred and that action is included in the determination of the final policy π. This formula captures the idea that is at the heart of the *expected value maximization* of decision theory:

$$BestDo([\delta_1|\delta_2]; rest, s, h, \pi, v, pr) \doteq$$
$$\exists \pi_1, v_1, pr_1. BestDo(\delta_1; rest, s, h, \pi_1, v_1, pr_1) \wedge$$
$$\exists \pi_2, v_2, pr_2. BestDo(\delta_2; rest, s, h, \pi_2, v_2, pr_2) \wedge$$
$$((v_1, \delta_1) \geq (v_2, \delta_2) \wedge \pi = \pi_1 \wedge v = v_1 \wedge pr = pr_1) \vee$$
$$((v_1, \delta_1) < (v_2, \delta_2) \wedge \pi = \pi_2 \wedge v = v_2 \wedge pr = pr_2)).$$

3. POMDP defined

3.1. The model

In partially observable Markov decision processes (POMDPs) actions have nondeterministic results and observations are uncertain. In other words, the effect of some chosen action is somewhat unpredictable, yet may be predicted with a probability of occurrence. And the world is not directly observable; some data are observable, and the agent infers how likely it is that the state of the world is in some specific state. The agent thus believes to some degree—for each possible state—that it is in that state, but it is never certain exactly which state it is in. Furthermore, a POMDP is a *decision* process and thus facilitates making decisions as to which actions to take, given its previous observations and actions.

Formally, a POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, b_0 \rangle$ with the following seven components (see e.g., [5, 6]): (1) $\mathcal{S} = \{s_0, s_1, \ldots, s_n\}$ is a finite set of states of the world; the state at time t is denoted s^t; (2) $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$ is a finite set of actions; (3) $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$ is the *state-transition function*, giving for each world state and agent action, a probability distribution over world states; (4) $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the *reward function*, giving the immediate reward that the agent can gain for any world state and agent action; (5) $\Omega = \{o_0, o_1, \ldots, o_m\}$ is a finite set of observations the agent can experience of its world; (6) $\mathcal{O} : \mathcal{S} \times \mathcal{A} \to \Pi(\Omega)$ is the *observation function*, giving for each agent action and the resulting world state, a probability distribution over observations; and (7) b_0 is the initial probability distribution over all world states in $\mathcal{S}$.

An important function is the function that updates the agent's belief: [5] call this function the *state estimation* function $SE(b, a, o)$. b is a set of pairs (s, p) where each state s is associated with a probability p, that is, b is a probability distribution over the set S of all states. b can be called a *belief state*. SE is define as

$$b^t(s') = \frac{\mathcal{O}(s', a, o) \sum_{s \in S} T(s, a, s') b^{t-1}(s)}{Pr(o|a, b)}, \qquad (1)$$

where $b^t(s')$ is the probability of the agent being in state s' at time-step t. (Action and observation subscripts have been ignored.) Equation (1) is derived from the Bayes Rule. $Pr(o|a, b)$ in the denominator is a normalizer; it is constant with time. SE returns a new belief distribution for every action-observation pair. SE captures the Markov assumption: a new state of belief depends only on the immediately previous observation, action and state of belief.

3.2. Determining a policy

For any set of sequences of actions, the sequence of actions that results in the highest expected reward is preferred. The *optimality prescription* of utility theory states: Maximize "the expected sum of rewards that [an agent] gets on the next k steps," [5].

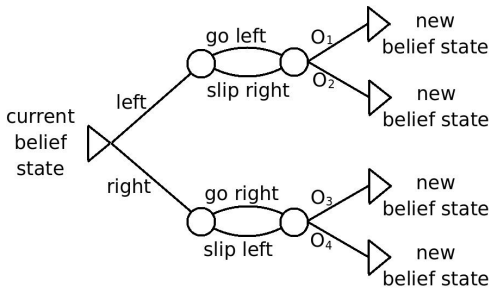

Figure 1: One tier of a POMDP-decision-tree.

That is, an agent should maximize $E\left[\sum_{t=0}^{k-1} r_t\right]$ where r_t is the reward received on time-step t.

When the states an agent can be in are belief states, we need a reward function over belief states. We derive $Rb(a, b)$ from the reward function over world states, such that a reward is proportional to the probability of being in a world state:

$$Rb(a, b) = \sum_{s \in S} R(a, s) \times b(s). \quad (2)$$

Now the aim of using POMDP models is to determine recommendations of 'good' actions or decisions. Such recommendations are called a *policy*. Formally, a policy (π) is a function from a set of belief states (all those the agent can be in) to a set of actions: $\pi : B \rightarrow A$. That is, actions are *conditioned* on beliefs. So given B_0, the first action a' is recommended by π. But what is the next belief state? This depends on the next observation. Therefore, for each observation associated with a', we need to consider a different belief state. Hence, the next action, a'', actually depends on the observations associated with (immediately after) a'. In this sense, a policy can be represented as a *policy tree*, with nodes being actions and branches being observations. The above equation is thus transformed to $\pi : \mathcal{O} \rightarrow A$. Now once we *have* a policy, it is independent of the agent's beliefs.

Let $V_{\pi,t}(s)$—the *value function*—be the expected sum of rewards gained from starting in world state s and executing policy π for t steps. If we define a value function over *belief* states as $Vb_{\pi,t}(b) = \sum_{s \in S} V_{\pi,t}(s) \times b(s)$, we can define the *optimal* policy $\pi*$ with planning horizon h (set $t = h$) as

$$\pi* = argmax_{\pi}(Vb_{\pi,h}(b_0)) \quad (3)$$

(from the initial belief state)—the policy that will advise the agent to perform actions (given any defined observation) such that the agent gains maximum rewards (after h actions).

To implement Equation (3), the authors make use of a decision tree (there are other methods). DTGolog uses a similar approach: *forward search value iteration*. An example sub-decision-tree (one tier) is shown in Figure 1. This example is based on an environment and agent model where the agent can only go left or right and each of its two actions has two possible realizations in the environment; also, the agent may make two kinds of observations (O_1 and O_2) if it chose to go left, and another two kinds of observations (O_3 and O_4) if it chose to go right.

Belief states (triangles) in the decision tree are decision nodes, that is, at these nodes, the agent can choose an action (make a decision). Circles are chance nodes, that is, certain events occur, each with a probability (chance) such that any one event at one chance node will definely happen (probabilities of branches leaving a chance node, sum to 1).

In Decision Analysis (see e.g., [7]), we roll back a decision tree to 'decide' the action. In any decision tree, for each action-observation pair, there is a tier of sub-decision-trees. That is, when considering n actions in a row, a decision tree with n tiers would be required. There is a unique path from the initial decision node to each leaf node, and at each belief state encountered on a path, a reward is added, until (and including) the leaf belief state. At this point, the agent knows the total reward the agent would get for reaching that final state of belief. Each of the belief states is reachable with some probability.

At each decision node, a choice is committed to. We iteratively roll back—from last decision nodes to first decision node. The agent can in this way decide at the first decision node, what action to take. Each subtree rooted at the end of the branches representing the agent's potential action, has an associated expected reward. The action rooted at the subtree with the highest expected reward, should be chosen.

As the decision tree is rolled back, the best decision/action is placed into the policy, conditioned on the most recent possible observations. Using such a *policy tree* (generated from a decision tree), the agent can always choose the appropriate action given its last observation. This is the essence of the theory on which our POMDP planner is based.

4. Related work

In the following, we present some related work dealing with reasoning under uncertainty. As there exists a large body of work in this field, we concentrate in particular on approaches for reasoning under uncertainty in the situation calculus and Golog.

[8]'s idea of representing beliefs is simple yet important. Intuitively, their aim is to represent an agent's uncertainty by having a notion of which configuration of situations are currently possible; the possible worlds framework. Then further, each possible world is given a likelihood weight. With these notions in place, they show how an agent can have a belief (a probability) about any sentence in any defined situation. Their work does not, however, cover planning.

Reiter [3] describes how to implement MDPs as well as POMDPs in the situation calculus. He defines the language *stGolog*, which stands for 'stochastic Golog'. Nevertheless, Reiter does not provide a method to automatically generate (optimal) policies, given a domain and optimization theory; he only provides the tools for the designer to program policies for partially observable decision domains by hand.

Grosskreutz shows how the Golog framework "can be extended to allow the projection of high-level plans interacting with noisy low-level processes, based on a probabilistic characterization of the robot's beliefs," [9]. He calls his extension to Golog *pGolog*. The belief update of a robot's epistemic state is also covered by [9]. (PO)MDPs are not employed in pGolog. Instead, he does probabilistic projection of specific programs. He does however make use of expected utility to decide between which of two or three or so programs to execute (after simulated scenarios).

In [11], Ferrein and Lakemeyer present the agent programming language *ReadyLog*. Approximately ten years after Golog's birth, ReadyLog combines many of the disparate useful features of the various dialects of Golog into one package. ReadyLog has been implemented and successfully used in robotic soccer competitions and a prototype domestic robot.

Whereas DTGolog [2] models MDPs—a useful model in

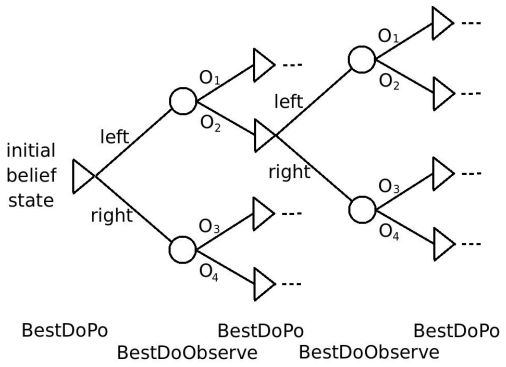

Figure 2: $BestDoPO$ represented as a POMDP-decision-tree.

robotics, as most robots operate in environments where actions have uncertain outcomes—our new dialect models *belief-MDPs*. A belief-MDP is one perspective of POMDPs, where the states that are being reasoned over are *belief* states and not the *world* states of MDPs. More detail concerning the semantics of DTGolog is given in Section 2.

Very related to our approach is the approach of [10]. Finzi and Lukasiewicz present a game-theoretic version of DTGolog to operate in partially observable domains. They call this extension *POGTGolog*. As far as we know, this is the only Golog dialect that can take partially observable problems as input, that is, that has some kind of POMDP solver for agent action planning. POGTGolog deals with multiple agents. Our work is different from theirs, as we concentrate on the single agent case and our agent is not restricted to game theory. For developers who prefer a Golog dialect for agent programming, but desire their robots or agents to operate with POMDP information, these developers cannot easily modify POGTGolog to work with single robots. Our work is not only a simplification of [10]; rather, we extend DTGolog, and use several elements in POGTGolog—either directly or for inspiration.

5. Semantics of POMDPs in Golog

In this section we describe our extension to the original forward search value iteration algorithm as proposed in [2]. In the following, we extend the approach of DTGolog in such a way that it can also deal with partially observable domains. In particular, instead of using $BestDo$, we introduce a predicate $BestDoPO$ to operate on a belief state rather than on a world state. $BestDoPO(p, b, h, \pi, v, pr)$ takes as arguments a Golog program p, a belief state b and a horizon h, which determines the solution depths of the algorithm. The policy π as well as its value v and the success probability are returned by the algorithm.

The relation of $BestDoPO$ to a POMDP-decision-tree can be seen in Figure 2. The stochastic outcomes of actions has been suppressed for ease of presentation.

An example of how $BestDoPO$ may be called initially—with a program that allows the agent to choose between three actions a_1, a_2, a_3 (without constraints), with b_0 the initial belief state and with the user or agent requiring advice for a sequence of seven actions—is $BestDoPO(\mathbf{while}\ true\ \mathbf{do}\ [a_1 \mid a_2 \mid a_3], b_0, 7, \pi, v)$.

5.1. Basic definitions and concepts

A belief state b contains the elements (s, p); each element/pair is a possible (situation calculus) situation s together with probability p (as in [10]).

We use the idea of [10] and assume that an action is possible in a belief state, when it it possible in the situation which is part of the belief state, that is, $Poss(a, b)$ iff $Poss(a, s)$. We add the predicate $PossObs(o, a, b)$ to the action theory, which specifies when an observation o is possible (perceivable) in belief state b, given an action a. We shall call $Poss(a, b)$, $PossAct(a, b)$, so that we can clearly distinguish between preconditions for observations and for actions. It is important to note that the b' in $PossObs(o, a, b')$ is the belief state reached *after* action a was executed. That is, if a was executed in b and b' is the new state reached, then $PossObs(o, a, b')$ says whether it is possible to observe o *after* a has been executed.

Next, we define a function symbol called $probNat(n, a, s)$ that is similar in meaning to the state transition function T of a Markov process. Our definition 'returns' a probability. It applies to all of nature's choices n, where s is the state in which the agent performs action a. Similarly, we introduce the function $probObs(o, a, s)$; the probability that o will be observed in s after a was executed in the previous situation.

Finally, we define $belObs$, which is the probability that the agent will observe some specified observation given its current beliefs and the sensor it activated: $belObs(o, a, b) \doteq \sum_{(s',p')\in b} p' \cdot probObs(o, a, s')$.

In the next section we briefly sketch our solution algorithm which calculates optimal policies under partial observability.

5.2. The partially observable $BestDo$

This subsection presents the key formulas in the definition of $BestDoPO$.

Considering possible observations after an action, we branch on all possible observations, given the robot's intended action a. $choiceObs'(a)$ 'returns' the set of observations that the robot may perceive: $\{o|choiceObs(o, a, s)$ for all $s \in S\}$. The reward function R is defined by (Eq. 2).

Probabilistic observation

$$
\begin{aligned}
&BestDoPO(a; rest, b, h, \pi, v, pr) \doteq \\
&\quad \neg PossAct(a, b) \wedge \pi = Stop \wedge v = 0 \wedge pr = 0 \ \vee \\
&\quad PossAct(a, b) \wedge \\
&\quad \exists \pi', v'. BestDoObserve(choiceObs'(a), \\
&\qquad a, rest, b, h, \pi', v', pr) \ \wedge \\
&\quad \pi = a; \pi' \wedge v = R(b) + v'.
\end{aligned}
$$

After a certain action a and a certain observation o_k, the next belief state is reached. At the time when the auxiliary procedure $BestDoObserve$ is called, a specific action, the set of nature's choices for that action and a specific observation associated with the action are under consideration. These elements are sufficient and necessary to update the agent's current beliefs. Inside $BestDoObserve$, the belief state (given a certain action and observation history) is updated via a belief state transition function (similar in vein to the state estimation function of Section 3, and the successor-state axiom for likelihood weights as given in [8]).

Belief update function

$b_{new} = BU(o, a, b) \doteq$

for each $(s, p) \in b$

$\exists n, s^+, p^+.(s^+, p^+) \in b_{temp} : s^+ = do(n, s) \wedge$

$choiceNat(n, a, s) \wedge PossAct(n, s) \wedge$

$p^+ = p \cdot probObs(o, a, s^+) \cdot probNat(n, a, s)$

end for each

$b_{new} = normalize(b_{temp})$.

A major difference between the POMDP model as defined in Section 3 and the POMDP model we define here for the situation calculus, is that here the belief state is not a probability distribution over a *fixed* set of states. If a situation (state) was part of the belief state to be updated, it is removed from the new belief state, and situations (states) that are 'accessible' from the removed situation via $choiceNat(n, a, s)$ and are executable via $PossAct(n, s)$ are added to the new belief state. Because non-executable actions result in situations being discarded, the 'probability' distribution over all the situations in the new belief state may not sum to 1; the distribution thus needs to be normalized.

$senseCond$ is mentioned in the definition of $BestDoObserve$: It is similar to the the definition in Section 2, only, here it is defined for observations instead of actions.

$BestDoPO$ is recursively called with the remaining program and with the horizon h decremented by 1. Also note that the recursive $BestDoPO$ will now operate with the updated belief b'. In the following definition, $\{o_k\}$ is a single (remaining) observation in the set returned by $choiceObs'$.

Observations possible

$BestDoObserve(\{o_k\}, a, rest, b, h, \pi, v, pr) \doteq$

$\neg PossObs(o_k, a, b) \wedge \pi = Stop \wedge v = 0 \wedge pr = 0 \vee$

$PossObs(o_k, a, b) \wedge b' = BU(o_k, a, b) \wedge$

$\exists \pi', v', pr'.BestDoPO(rest, b', h-1, \pi', v', pr') \wedge$

$senseCond(o_k, \varphi_k) \wedge \pi = \varphi_k?; \pi' \wedge$

$v = v' \cdot belObs(o_k, a, b) \wedge pr = pr' \cdot belObs(o_k, a, b)$.

When the set of observations has more than one observation in it, the formula definition is slightly different, but similar to the one above: the first branch of possible observations is processed, and the other branches in the remainder of the set are processed recursively.

When the planning horizon has reached zero or when all actions have been 'performed' (no remaining actions in the input program), there will be no further recursive calls.

Conditional statement and test action formulas are similar to those of Golog, except that the 'condition' or 'test statement' respectively, are with respect to the agent's current belief state, and probabilities involved in these formulas are influenced in proportion to the agent's *degree of belief* [8] in the respective statements (see [10] for details). Sequential composition and conditional iteration are defined as one would expect according to complex actions in Golog.

6. A Simple Example

A very simple example follows to illustrate how $BestDoPO$ calculates an optimal policy. We use a four-state world as depicted in Figure 3. The agent's initial belief state is $B_0 = \{(s1, 0.04), (s2, 0.95), (s3, 0.0), (s4, 0.01)\}$. The only actions available to the agent are $left$ and $right$. We define the actions' stochasticity with $\forall n, a, s.choiceNat(n, a, s) \equiv TRUE$, with associated probabilities:

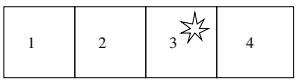

Figure 3: Four-state world; four states in a row. Initially the agent believes it is in each state with probabilities [0.04|0.95|0.00|0.01] corresponding to state position.

$probNat(left, left, s) = probNat(right, right, s) = 0.9$

$probNat(right, left, s) = probNat(left, right, s) = 0.1$

The probability that any of the actions will cause an observation of nothing ($obsnil$) is 1: $probObs(obsnil, a, s) = p \equiv (a = left \vee a = right) \wedge p = 1$. The corresponding definition for choice of observations is $choiceObs(obsnil, a, s) \equiv (a = left \vee a = right)$.

Let the fluent $At(loc(x), s)$ denote the location of the agent. It's successor-state axiom is defined by

$$\begin{aligned} &At(loc(x), do(a, s)) \equiv \\ &\quad a = left \wedge (At(loc(x+1), s) \wedge x \neq 1) \\ &\qquad \vee (At(loc(x), s) \wedge x = 1) \vee \\ &\quad a = right \wedge (At(loc(x-1), s) \wedge x \neq 4) \\ &\qquad \vee (At(loc(x), s) \wedge x = 4) \vee \\ &\quad At(loc(x), s) \wedge (a \neq left \wedge a \neq right). \end{aligned}$$

For simplicity, we allow all actions and observations all of the time, that is, $\forall a, s.PossAct(a, s) \equiv TRUE$ and $\forall a, s.PossObs(a, s) \equiv TRUE$.

Finally, we specify the sensing condition predicate and the reward function. $senseCond(obsnil, \psi) \equiv \psi = OutcomeIs(nil, sensor\_value))$ with $OutcomeIs(obsnil, sensor\_value) \equiv TRUE$ and $reward(s) =$ **if** $At(loc(3), s)$ **then** 1 **else** -1; hence, the agent's goal should be location 3.

Assume, the agent is equipped with the following program; an initial input for $BestDoPO$:

$BestDoPO(\textbf{while}\ (true\ \textbf{do}\ [left \mid right]),$

$\{(s1, 0.04), (s2, 0.95), (s3, 0.0), (s4, 0.01)\}, 1, \pi, v, pr)$;

the algorithm must computing a one-step optimal policy.

After the iterative component of the program is processed, the following call is made, as per the definition of $BestDoPO$ for the nondeterministic choice of actions:

$BestDoPO([left \mid right]; rest, B_0, 1, \pi, v, pr)$

$\exists \pi_1, v_1, pr_1.BestDoPO(left; rest, b, 1, \pi_1, v_1, pr_1) \wedge$

$\exists \pi_2, v_2, pr_2.BestDoPO(right; rest, b, 1, \pi_2, v_2, pr_2) \wedge$

$((v_1, left) \geq (v_2, right) \wedge \pi = \pi_1 \wedge v = v_1 \wedge pr = pr_1) \vee$

$(v_1, left) < (v_2, right) \wedge \pi = \pi_2 \wedge v = v_2 \wedge pr = pr_2))$,

where $rest$ is **while** $(true\ \textbf{do}\ [left \mid right])$. Then the recursive $BestDoPO$s make use of the "Probabilistic observation" definition of the formula. Because–by the action precondition axioms for this example–$left$ and $right$ are always executable, the following portion (times two) of the formula are applicable:

$$\exists \pi', v'.BestDoObserve(choiceObs'(left), \tag{4}$$

$$left, rest, B_0, 1, \pi', v', pr) \wedge \tag{5}$$

$$\pi = left; \pi' \wedge v = R(B_0) + v' \tag{6}$$

and

$$\exists \pi', v'.BestDoObserve(choiceObs'(right), \quad (7)$$

$$right, rest, B_0, 1, \pi', v', pr) \wedge \quad (8)$$

$$\pi = right; \pi' \wedge v = R(B_0) + v'. \quad (9)$$

For Lines (4) and (5) the following portion of the "Observations possible" definition is applicable:

$$b' = BU(obsnil, left, B_0) \wedge$$
$$\exists \pi', v', pr'.BestDoPO(rest, b', 1-1, \pi', v', pr') \wedge$$
$$senseCond(obsnil, \phi) \wedge \pi = \phi?; \pi' \wedge$$
$$v = v' \cdot belObs(obsnil, left, B_0) \wedge$$
$$pr = pr' \cdot belObs(obsnil, left, B_0).$$

In this formula (portion), ϕ unifies with $OutcomeIs(obsnil,sensor\_value)$ and because the recursive call to $BestDoPO$ has a zero horizon, $\pi' = nil$, and thus $\pi = (OutcomeIs(obsnil,sensor\_value))?;nil$.

The updated belief is an input to a 'zero horizon' call and will therefore be used to determine v'; we calculate the new belief state, $b' = BU(obsnil, left, B_0)$, now. We work out only the first new element of b' in detail:

$$(s^+, p^+) \in b_{temp} : s^+ = do(left, s_1) \wedge\ p^+ = 0.04 \times 1 \times 0.9.$$

Because all actions are possible, the only effect that normalization (in the update function) has, is to remove $(do(left, s3), 0.0)$ and $(do(right, s3), 0.0)$ from the new belief state, because of their zero probabilities. $BU(obsnil, left, B_0)$ results in

$$b' = \{(do(left, s1), 0.036), (do(right, s1), 0.004),$$
$$(do(left, s2), 0.855), (do(right, s2), 0.095),$$
$$(do(left, s4), 0.009), (do(right, s4), 0.001)\}.$$

$belObs(obsnil, left, B_0) = (0.04)(1) + (0.95)(1) + (0.0)(1) + (0.01)(1) = 1$ and hence $v = v' \times 1$, and $pr = pr' \times 1$. Due to the 'zero horizon' call, $v' = R(b') = (-1)(.036)+(-1)(.004)+(-1)(.885)+(1)(.095)+(1)(.009)+(-1)(.001) = -0.822$ and $pr' = 1.0$. Therefore, $v = -0.822$, and $pr = 1.0$.

Now we can instantiate Line (6) as follows: $\pi = left; OutcomeIs(obsnil, sensor\_value))?; nil \wedge v = -1 + (-0.822)$. Similarly, we can instantiate Line (9) as $\pi = right; OutcomeIs(obsnil, sensor\_value))?; nil \wedge v = -1 + (0.712)$.

Then finally, we find that $((-0.822, left) < (0.712, right)$ and return the policy $\pi = right; OutcomeIs(obsnil, sensor\_value))?; nil$, with total expected reward $v = -0.288$ and program success probability $pr = 1$.

Note that for the sake of clarity, we assumed noise-free perceptions. It should be clear though, that our algorithm can deal with noisy perceptions as well.

Considering that the agent believed to a relatively high degree that it was initially just left of the 'high-reward' location, and given that its observations are complete and its actions are not extremely erroneous, we would expect the agent's first move to be rightwards, as indeed, the policy recommends.

7. Discussion and Conclusion

In this paper we have given a formal semantics for an action planner that can generate control policies for agents in partially observable domains. The language we used for the specification is the agent programming language DTGolog. Much of the semantics is similar to [10]. Their approach is however not for a single-agent domain.

An example was presented that showed in detail the processes involved in generating a policy for an agent with probabilistic beliefs in a partially observable and stochastic domain.

We implemented the POMDP planner in ECL^iPS^e *Prolog*. The implementation was set up for two toy worlds: a four-state world where the states are all in a row, and a five-by-five grid world. In both cases, an agent must find a 'star'. Preliminary experiments with the implementation showed the potential for practical application of the planner presented in this paper: the results of the experiments showed that the policies generated are reasonable, and overall, the planner seems to work correctly. However, benchmarking and comparison to other similar planners (for problems in similarly stochastic and noisy domains) still needs to be conducted.

8. References

[1] Levesque, H., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R., "GOLOG: A Logic programming language for dynamic domain", *Journal of Logic Programming*, 31:59–84, 1997.

[2] Boutilier, C., Reiter, R., Soutchanski, M., and Thrun, S., "Decision-theoretic, high-level agent programming in the situation calculus", in *Proceedings AAAI-2000*, 2000, pp. 355–362.

[3] Reiter, R., *Knowledge in action: logical foundations for specifying and implementing dynamical systems*, Massachusetts/England: MIT Press, 2001.

[4] Brachman, R. J. and Levesque, H. J., *Knowledge representation and reasoning*, California: Morgan Kaufmann, 2004.

[5] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R., "Planning and acting in partially observable stochastic domains", *Artificial Intelligence*, 101(1–2):99–134, 1998.

[6] Pineau, J., *Tractable planning under uncertainty: exploiting structure*, Robotics Institute, Carnegie Mellon University, 2004. Unpublished doctoral dissertation.

[7] Clemen, R. T., and Reilly, T., *Making hard decisions*, California: Duxbury, 2001.

[8] Bacchus, F., Halpern, J. Y., and Levesque, H. J., "Reasoning about noisy sensors and effectors in the situation calculus", *Artificial Intelligence*, 111(1–2):171–208, 1999.

[9] Grosskreutz, H., *Towards more realistic logic-based robot controllers in the Golog framework*, Knowledge-Based Systems Group, Rheinisch-Westfälischen Technischen Hochschule, 2002.

[10] Finzi, A, and Lukasiewicz, T., "Game-theoretic agent programming in Golog under partial observability", in *KI 2006: Advances in Artificial Intelligence*, 2007, pp. 113–127.

[11] Ferrein, A. and Lakemeyer G. "Logic-based robot control in highly dynamic domains.", *Journal of Robotics and Autonomous Systems, Special Issue on Semantic Knowledge in Robotics* 2008. to appear.

Appendix D

Paper 2 – A BDI Agent Architecture for a POMDP Planner

Conference paper [86] resulting from work in this dissertation.

G. Rens, A. Ferrein, and E. van der Poel. A BDI agent architecture for a POMDP planner. In G. Lakemeyer, L. Morgenstern, and M-A. Williams, editors, *Proc. of 9th Intl. Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense 2009)*, pages 109–114, University of Technology, Sydney, 2009. UTSe Press.

The paper starts on the next page.

A BDI Agent Architecture for a POMDP Planner

Gavin Rens[1,2]
Alexander Ferrein[3]
Etienne van der Poel[1]
[1] School of Computing, Unisa, Pretoria, South Africa
[2] Knowledge Systems Group, Meraka Institute, CSIR, Pretoria, South Africa
[3] Robotics and Agents Research Laboratory, University of Cape Town, South Africa
grens@csir.co.za, alexander.ferrein@uct.ac.za, evdpoel@unisa.ac.za

Abstract

Traditionally, agent architectures based on the Belief-Desire-Intention (BDI) model make use of *pre-compiled* plans, or if they do *generate* plans, the plans do not involve stochastic actions nor probabilistic observations. Plans that *do* involve these kinds of actions and observations are generated by partially observable Markov decision process (POMDP) planners. In particular for POMDP planning, we make use of a POMDP planner which is implemented in the robot programming and plan language Golog. Golog is very suitable for integrating beliefs, as it is based on the situation calculus and we can draw upon previous research on this. However, a POMDP planner on its own cannot cope well with dynamically changing environments and complicated goals. This is exactly a strength of the BDI model; the model is for reasoning over goals dynamically. Therefore, in this paper, we propose an architecture that will lay the groundwork for architectures that combine the advantages of a POMDP planner written in the situation calculus, and the BDI model of agency. We show preliminary results which can be seen as a proof of concept for integrating a POMDP into a BDI architecture.

Introduction

Traditionally, plan-based agents that include generative planning (as opposed to utilizing pre-compiled plans) would generate a complete plan to reach a *specific fixed* goal, then execute the plan. If plan execution monitoring is available, the agent would replan from scratch when the plan becomes invalid. Due to the time requirements for generating complete plans, the plan may be invalid by the time it is executed. This is because the world may change substantially during plan generation.

Therefore, Belief-Desire-Intention (BDI) architectures take a different approach. BDI theory is based on the philosophy of practical reasoning (Bratman 1987). It offers flexibility in planning beyond traditional planning for agents, by reasoning over *different goals*. That is, an agent based on BDI theory can adapt to changing situations by focusing on the pursuit of the most appropriate goal at the time. Typically, an appropriate plan to achieve an adopted goal is then selected from a data base of plans. Although a plan that satisfies certain constraints (e.g., does not conflict with other adopted plans, is executable, etc.) will be adopted, it may not be the most appropriate plan in existence. A plan that is generated with the agent's current knowledge for guidance, may be more appropriate. BDI agents can also make rational decisions as to when to replan if a plan becomes invalid, reducing the amount of replanning, thus increasing the agent's reactivity. Note that the BDI model is, however, not the only approach to replanning (cf. (Likhachev et al. 2005)).

In general, BDI architectures do not make use of plan generation, they rather draw on plan libraries. While with BDI approaches, an agent can reason over several goals, the agent lacks some flexibility by not being able to generate suitable plans on demand. Therefore, in this paper, we aim at integrating a POMDP planner into a BDI architecture to combine its benefits with the ability to *generate* plans. Moreover, we want to supply models that are as realistic as possible. We therefore decided on employing partially observable Markov Decision Processes (POMDPs).

In this paper we describe our approach for combining BDI theory with a POMDP planner. Combining the two formalisms can be viewed from two perspectives. One, to enhance an existing planner for use in real-time dynamic domains by incorporating the planner into a BDI agent architecture so that the management of goal selection, planning and replanning is handled in a principled way. Two, to enhance the classical BDI agent architecture by incorporating a POMDP planner into the BDI architecture so that the agent can reason (plan) with knowledge about the uncertainty of the results of its actions, and about the uncertainty of the accuracy of its perceptions. We employ the POMDP planner described in our previous work (Rens, Ferrein, and Van der Poel 2008). This planner is implemented in Golog (Levesque et al. 1997), which in turn is based on the situation calculus (McCarthy 1963; Reiter 2001). An advantage of using a Golog implementation for the planner is that the integration of beliefs into the situation calculus has previously been done (e.g., (Bacchus, Halpern, and Levesque 1999)) and this work can be used for formulating POMDPs. Further, given a background action theory, an initial state and a goal state (or reward function in POMDPs), Golog programs essentially constrain and specify the search space (the space of available actions).

The resulting plan (or *policy* in POMDPs) is a Golog program which can be executed directly by the agent. To the

best of our knowledge, till present, no BDI-based agent architecture has implemented its planning function so as to generate plans that take stochastic action and partial observation into account. Therefore, this work can be seen as a first proof of this concept.

The rest of the paper is organized as follows. In the next section we introduce the plan generator used in this study. Then, we briefly introduce the BDI theory, after which we explain our hybrid BDI/POMDP-planner architecture in detail. Before we conclude, we show some preliminary results from an implementation of our architecture, which gives a first proof of our approach.

The Planning Module

The POMDP Model

In partially observable Markov decision processes (POMDPs) actions have nondeterministic results, yet may be predicted with a probability of occurrence. And observations are uncertain: the world is not directly observable, therefore the agent *infers* how likely it is that the world is in some specific state. The agent thus believes to some degree—for each possible state—that it is in that state. Furthermore, a POMDP is a *decision* process and thus facilitates making decisions as to which actions to take, given its previous observations and actions. Formally, a POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, b_0 \rangle$ with: $\mathcal{S}$, a finite set of states of the world; $\mathcal{A}$, a finite set of actions; $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$ is the *state-transition function*, where Π is a probability distribution; $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, the *reward function*; Ω, a finite set of observations the agent can experience; $\mathcal{O} : \mathcal{S} \times \mathcal{A} \to \Pi(\Omega)$, the *observation function*; and b_0, the initial probability distribution over all world states in $\mathcal{S}$ (see e.g., (Kaelbling, Littman, and Cassandra 1998)). In the model, b is a *belief state*, i.e. a set of pairs (s, p) where each state $s \in \mathcal{S}$ is associated with a probability p. The *state estimation* function $SE(b, a, o)$ updates the agent's beliefs. Now the aim of the agent deploying a POMDP model is to determine a policy, that is, the actions or decisions that will maximize its rewards. Formally, a policy π is a function from a set of belief states B to the set of actions: $\pi : B \to A$. That is, actions are *conditioned* on beliefs. This means that the agent takes its next decision not only based on a stochastic action model, but also on a stochastic observation model. In this sense, a policy can be represented as a *policy tree*, with nodes being actions and branches being observations.

Planning over Degrees of Belief

In this section we describe our POMDP planner, an extension to the decision-theoretic language, DTGolog (Boutilier et al. 2000).

DTGolog is based on Reiter's variant of the situation calculus (McCarthy 1963; Reiter 2001), a second-order language for reasoning about actions and their effects. According to this calculus, changes in the world are due only to actions, so that a situation is completely described by the history of actions starting in some initial situation—$do(a, s)$ is the term denoting the situation resulting from doing action a in situation s. Properties of the world are described by *fluents*, which are situation-dependent predicates and functions. For each fluent the user defines a successor state axiom specifying precisely which value the fluent takes on after performing an action. These, together with precondition axioms for each action, axioms for the initial situation, and foundational and unique names axioms, form a so-called *basic action theory* (Reiter 2001).

Decision-theoretic planning in DTGolog works roughly as follows. Given an input program that leaves open several action alternatives for the agent, the DTGolog interpreter generates an optimal policy. Formally, the interpreter solves a Markov Decision Process (MDP, cf. e.g., (Puterman 1994)) using the forward search value iteration method—searching (to a specified horizon) for the actions that will maximize the total expected reward. Programs are interpreted as follows: All possible outcomes of the intended nondeterministic, stochastic action are expanded. For each choice point, the action resulting in the optimal value at the particular point in the MDP, is determined. These values are calculated relative to the world situation associated with the point in the MDP. The policy is calculated with an optimization theory consisting of a reward and a transition function (cf. also (Boutilier et al. 2000)). The transition function describing transition probabilities between states of the Markov chain is given by Reiter's variant of the basic action theory formalized in the underlying situation calculus (McCarthy 1963; Reiter 2001). Formally, the $BestDo$ macro defines the process described above: it evaluates an input program and recursively builds an optimal policy.

The POMDP planner we use here is $BestDoPO$ (Rens, Ferrein, and Van der Poel 2008); an extension of $BestDo$ ($BestDo$ Partially Observable), which calculates an optimal policy for the partially observable case (Rens, Ferrein, and Van der Poel 2008). The main difference is that $BestDoPO$ operates on a belief state rather than on a world state. $BestDoPO(p, b, h, \pi, v, pr)$ takes as arguments a Golog program p, a belief state b and a horizon h, which determines the solution depth sought by the interpreter. The policy π as well as its value v and the success probability pr are returned. After a certain action a is performed and the associated observation o is perceived, the next belief state is determined via a belief state transition function (similar in vein to the state estimation function of the previous subsection, and the successor-state axiom for likelihood weights as given in (Bacchus, Halpern, and Levesque 1999)):

$$
\begin{aligned}
&b_{new} = BU(o, a, b) \doteq \\
&\quad b_{temp} = \{(s^+, p^+) \mid (\exists n, s^+, p^+).(s^+, p^+) \in b_{temp} : \\
&\qquad s^+ = do(n, s) \wedge choiceNat(n, a, s) \wedge PossAct(n, s) \wedge \\
&\qquad p^+ = p \cdot probObs(o, a, s^+) \cdot probNat(n, a, s)\} \\
&\quad b_{new} = normalize(b_{temp}).
\end{aligned}
$$

$choiceNat(n, a, s)$ specifies the possible outcomes n of the agent's intention to perform action a. $PossAct(n, s)$ denotes the possibility of performing action n in situation s. $probObs(o, a, s^+)$ and $probNat(n, a, s)$ are functions that

return the probability of observing o in the situation s^+—the situation resulting from doing action a, and respectively, the probability of action n being the outcome of the intention to execute action a in situation s.

For $BestDoPO$ to be integrated as required for the present work, two arguments are added to the list: $BestDoPO$ as defined in (Rens, Ferrein, and Van der Poel 2008) is modified to return δ and to take nom. The input program may provide information for a sequence of actions of length greater than the policy horizon. Call the remaining program δ—the portion of the program that was not used for policy generation. δ becomes the new program from which future policies will be generated. nom—the name of the input program—is used to select the reward function associated with the input program. Two clauses that are part of the definition of the modified $BestDoPO$ appear below.

$$
\begin{aligned}
&BestDoPO(p,\delta,nom,b,h,\pi,v,pr) \stackrel{def}{=} \\
&\quad h=0 \wedge \delta = p \wedge \\
&\quad \pi = stop \wedge \exists v.believedReward(nom,v,b) \wedge pr = 1. \\
&BestDoPO(a:p,\delta,nom,b,h,\pi,v,pr) \stackrel{def}{=} \\
&\quad \neg actionBelievedPossible(a,b) \wedge \\
&\qquad \delta = p \wedge \pi = stop \wedge v = 0 \wedge pr = 0 \vee \\
&\quad actionBelievedPossible(a,b) \wedge \\
&\qquad \exists\, obs.setofAssocObservations(a,obs) \wedge \\
&\qquad \exists \pi', v', pr.Aux(obs,a,p,\delta,nom,b,h,\pi',v',pr) \wedge \\
&\qquad believedReward(nom,r,b) \wedge \pi = a;\pi' \wedge v = r + v'.
\end{aligned}
$$

Please refer to (Rens, Ferrein, and Van der Poel 2008) for more detail.

BDI Theory

A desire is understood as what an agent ideally wants to achieve, that is, what motivates it. In reality, agents are resource-bounded, and hence should rationally choose the desires to pursue whose achievement are most valuable to the agent and that are achievable according to the agent's current situation and capabilities. The desires that have been committed to pursuing through a rational process of reasoning may be called *intentions*. The Belief-Desire-Intention (BDI) model of agency takes intentions—in addition to beliefs and desires—as first-class mental states. Traditional agent architectures either simply do not consider intentions, or do not consider them as explicit operands within the processes of an agent's reasoning system.

The value of taking intentions seriously is that they manage the agent's resources in a rational way. Intentions induce the agent to act and intentions persist. As such, they focus the agent's activity to commit resources and thus pursue a desire more effectively. Also, because intentions persist, new intentions are not constantly being adopted: new intentions are constrained by current intentions, and hence, future deliberation is constrained (Wooldridge 2000).

It is useful to distinguish between *deliberation*: to decide on what ends (e.g., reward functions; goal states) to pursue and *means-ends reasoning*: how to achieve the ends. Deliberation may be further divided into (i) reasoning to generate options from beliefs, i.e., 'wishing' to decide on current desires; (ii) reasoning to select intentions, i.e., 'focusing' on a subset of those desires and committing to achieve them. Committed-to goals, or plans for achieving them, are *intentions*.

A BDI agent has at least these seven components (Wooldridge 1999):

- A knowledge base of beliefs.
- An option generation function ($wish$), generating the options the agent would ideally like to pursue (its desires).
- A set of desires $Dess$ returned by the $wish$ function.
- A function ($focus$) that filters out incompatible, impossible and less valuable desires, and that focuses on a subset of the desire set.
- A structure of intentions $Ints$—the most desireable options/desires returned by the $focus$ function.
- A belief change function ($update$): given the agent's current beliefs and the latest percept sensed, the belief change function returns the updated beliefs of the agent.
- A function (*execute*) that selects some action(s) from the plan the agent is currently executing, and executes the action(s).

In most of the well known implementations of agents based on the BDI model (e.g., PRS (Georgeff and Ingrand 1989), IRMA (Bratman, Israel, and Pollack 1988) and dMARS (Rao and Georgeff 1995)), the $plan$ function returns plans from a plan library; a set of pre-compiled plans. An *intention structure* then structures various plans into larger hierarchies of plans. An intention in the intention structure in the classical BDI theory is a partial plan structured as a hierarchy of subplans. Furthermore, subplans may at some point be abstract, waiting to be 'filled in' (Bratman, Israel, and Pollack 1988). Some BDI architectures are designed to let the $plan$ function generate plans from atomic actions (Sardina, De Silva, and Padgham 2006; Walczak et al. 2007) (or it may possibly use a combination of pre-compiled and generated plans). However, none of the architectures that have a generative component employ a planner that produces plans for a POMDP model.

Combining POMDP Planning with the BDI Model

In this section we see how an agent controller in the BDI model can incorporate the $BestDoPO$ POMDP planner into its practical reasoning processes. We took the prototypical control loop of the BDI model as a reference and modified it to accommodate planning with POMDP policies. The proposed architecture is called BDI-POP (BDI with POmdp Planner).

First we introduce some terms and their relationships with the aid of Figure 1 (next page). Implicitly included in the "BELIEF" data store, is a fixed set of behaviors $behs$ and a fixed set of reward functions $rwds$ ($rwds$ is considered globally accessible). $behs$ is the agent's primitive goals; its innate drive. The idea is that each behavior refers to a unique goal that the agent is designed to achieve. Each behavior is *defined* by the set of programs and reward functions that can achieve the behavior. The $wish$ function is

omitted from our architecture (for now) because the options the agent would pursue at any time are its behaviors $behs$. The agent also has a fixed set of desires d. Each $des \in d$ is a triple $(nom, prog, ach)$: nom is a reference to the Golog program $prog$, and ach is a reference to the behavior $beh \in behs$ that $prog$ can potentially achieve, thus $ach \in behs$. The reward functions $rf \in rwds$ take as argument a nom that refers to the program that rf is associated with. The following holds: $\forall beh.[beh \in behs \rightarrow (\exists des).des = (nom, prog, ach) \wedge ach = beh]$: for each behavior, there exists at least one program to achieve it.

To understand the controller, we also need to consider the agent's deliberation process. $deliberate$ is the procedure that calls and controls the $focus$ predicate and that operates on the intention stack. We write $focus(b, d, i, behs, h^-)$ to be the predicate that selects one $des \in d$ for each $beh \in behs$, placing these desires in a stack, in ascending order, ordered by the desires' values. The desire selected for a behavior is the one that can achieve the behavior ($ach = beh$) and that has the highest value. A desire's value is estimated as the value v of the policy found, generated to a depth h^-: $BestDoPO$ is called with b, h^- and the applicable $prog$ as arguments; v is used and the policy is discarded. We keep $h^- < h$ to save on time spent deliberating. $focus$ 'returns' the stack i of selected desires.

$$
\begin{aligned}
&deliberate(b, d, i, behs, ai, i', h^-) \stackrel{def}{=} \\
&\quad (isEmpty(i) \wedge \exists i'.focus(b, d, i, behs, h^-) \vee \\
&\quad \neg isEmpty(i) \wedge \exists i'.i' = i) \wedge \\
&\quad \exists ai, i''.popIntentionStack(i', ai, i'').
\end{aligned}
$$

BDI-POP tests whether a usable policy could be generated, that is, whether the planner returns the $stop$ policy: When every outcome of an intended action (according to the input program) is illegal (according to the background action theory), $BestDoPO$ returns $stop$, and we say that the input program is *impossible*. An intention $i = (nom, prog, ach)$ with $prog$ being impossible is thus defined as an *impossible intention*.

The strategy used in $deliberate$ to deal with an impossible intention is extremely simple: it is dropped and the next intention on the stack is popped. This is a reasonable strategy because the next intention on the stack has the highest value, and should thus be pursued next. Calling $focus$ to refill the intention stack at this time would defeat the principle of *commitment* to intentions. Other strategies are possible, for example, replacing the impossible intention with another intention that achieves the same behavior, if one exists.

A logical high-level specification of BDI-POP follows, after which, it is explained in words.

$$
\begin{aligned}
&Agent(b, d, i, behs, ai, \pi, h, h^-) \stackrel{def}{=} \\
&\quad (nom, p, ach) = ai \wedge \\
&\quad \pi \neq stop \wedge p \neq nil \wedge \\
&\quad \exists \pi''.\pi = a; \pi'' \wedge execute(a) \wedge \\
&\quad \exists sv.getPercep(a, sv) \wedge \exists o.recognize(a, o) \wedge \\
&\quad \exists \pi'''.getSubPolicy(\pi'', o, \pi''') \wedge \\
&\quad \exists b'.b' = BU(o, a, b)) \wedge \\
&\quad Agent(b', d, i, behs, ai, \pi''', h, h^-).
\end{aligned}
$$

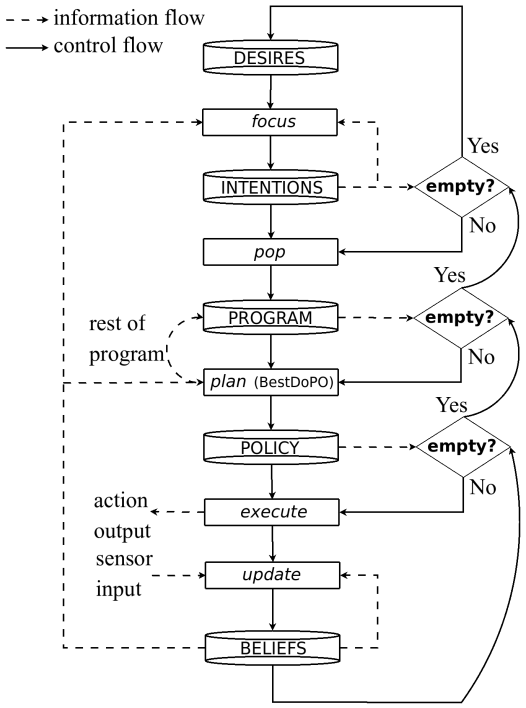

Figure 1: Schematic diagram of a sketch of the BDI architecture with the POMDP planner.

$$
\begin{aligned}
&Agent(b, d, i, behs, ai, \pi, h, h^-) \stackrel{def}{=} \\
&\quad (nom, p, ach) = ai \wedge \\
&\quad \pi = stop \wedge p = nil \wedge \\
&\quad deliberate(b, d, i, behs, ai', i', h^-) \wedge \\
&\quad Agent(b, d, i', behs, ai', \pi, h, h^-).
\end{aligned}
$$

$$
\begin{aligned}
&Agent(b, d, i, behs, ai, \pi, h, h^-) \stackrel{def}{=} \\
&\quad (nom, p, ach) = ai \wedge \\
&\quad \pi = stop \wedge p \neq nil \wedge \\
&\quad \exists \delta, \pi', v, pr.BestDoPO(p, \delta, nom, b, h, \pi', v, pr) \wedge \\
&\quad ai' = (nom, \delta, ach) \wedge \\
&\quad (\pi' = stop \wedge \\
&\qquad \exists ai'', i'.deliberate(b, d, i, behs, ai'', i', h^-) \wedge \\
&\qquad Agent(b, d, i', behs, ai'', \pi', h, h^-) \vee \\
&\quad \pi' \neq stop \wedge \\
&\qquad \exists \pi''.\pi' = a; \pi'' \wedge execute(a) \wedge \\
&\qquad \exists sv.getPercep(a, sv) \wedge \exists o.recognize(a, o) \wedge \\
&\qquad \exists \pi'''.getSubPolicy(\pi'', o, \pi''') \wedge \\
&\qquad \exists b'.b' = BU(o, a, b)) \wedge \\
&\qquad Agent(b', d, i, behs, ai', \pi''', h, h^-)).
\end{aligned}
$$

The agent follows the intention with the highest value—the intention popped from the stack. Call this the *active intention*. Initially, the intention stack is empty, so $deliberate$ is called and the active intention is instantiated. Whenever the controller needs a new plan to execute, $BestDoPO$ is

called to generate a policy with horizon h using the program specified by the active intention. The agent executes the policy until the end of the policy is reached, then $BestDoPO$ is called again for the rest of the program. If there is no rest of program (the program is empty), $deliberate$ is called. If the program has become impossible, $deliberate$ is called.

$getPercept$ returns a sensor value, given the action executed / sensor activated. The agent processes the sensor data and decides what it observed—the agent recognizes the sensor reading via the $recognize$ predicate, which outputs an observation. With this observation, the correct subpolicy is extracted from the current policy, and this (possibly empty) subpolicy becomes the new current policy.

After the action recommended by the policy is executed, the agent's beliefs must be updated according to what it 'knows' about the effects of its actions. The same belief update function used during planning by $BestDoPO$ is used to update the agent's beliefs. The current belief state of the agent will be the 'initial' belief state required as argument to $BestDoPO$ the next time the planner is called.

Given our present definition of $deliberate$ and given that we shall allow only finite programs for achieving intentions, the agent is guaranteed to deliberate at regular intervals. However, this interval period is fixed (to the degree that intentions become impossible). Adding a $reconsider$ predicate that tells the agent once every control cycle whether to deliberate, is a more sophisticated method. $reconsider$ is described by, for example, Wooldridge (2000) and "was examined by David Kinny and Michael Georgeff, in a number of experiments," (Wooldridge 1999, p. 57). Because we are investigating the feasibility of the basic idea of the hybrid architecture in this paper, we have left out the $reconsider$ predicate from the present investigation.

A somewhat significant difference of our hybrid architecture from the perspective of control via POMDP policy generation, is that—as stand-alone controller—the POMDP planner takes a single plan with a single associated reward function, to generate a policy. The new hybrid architecture takes several programs, each with an associated reward function. This aspect of the agent being able to reason over multiple behaviors has the advantage that the agent designer can separately specify behaviors that should—at least intuitively—be considered separately.

$BestDoPO$ expands Golog programs into hierarchically structured plans (policies), and only programs that have been selected as intentions are expanded into policies. Each program can generate a policy—or several policies if the program is expanded piece-wise. Viewing a policy tree as an intention structure in the sense of traditional BDI architectures, each program in the intention stack represents (at least one) intention structure. BDI-POP, therefore, maintains *several* unexpanded intention structures, only expanded when popped from the intention stack.

Implementation and First Experiments

To validate the BDI-POP architecture and to gain a sense for its performance potential, we observe one agent based on the architecture, in a simulation. The simulation environment is inspired by Tileworld (Pollack and Ringuette 1990), a testbed for agents. We designed and implemented the FireEater world, a dynamically changing grid world (a 5×5, two dimensional grid of cells) in which our agent is situated. There are obstacles that change position and fires that can be 'eaten'. Space prohibits a detailed explanation of FireEater world.

The agent gets one 'fire-point' for eating one fire. It can only eat a fire if it is in the same cell as the fire. There are two agent behaviors: $\texttt{findFood}, \texttt{eat} \in behs$. `findFood` may be realized by two available programs, and `eat` is forced to be achieved by one (other) program. The agent can go left, right, up or down—locomotive actions which are stochastically nondeterministic; it can sense its location (probabilistically) and it can eat fire (deterministically).

In order to have a base-line against which the performance of the new hybrid architecture can be compared, a simple or 'naive' architecture (called Naive-POP) was implemented. It has no explicit intentions or desires as defined for the BDI model. The agent is provided with a single Golog program and associated reward function. In this implementation, the program loops continuously over a nondeterministic action—nondeterministic between all available actions. If there is no rest of program, that is, the agent has executed the whole program, the agent will stop its activity.

BDI-POP, in contrast, does not employ programs that loop infinitely (in the experiments): programs were designed so that they become empty as soon as a policy is generated from the program. Hence an intention will be regarded as 'achieved' as soon as its policy becomes empty. Then the next intention will be popped from the stack. Because the size of the stack equals $|behs|$ and because all programs are finite, it is guaranteed that periodically all intentions will be achieved, that is, the stack is empty, and the agent is forced to deliberate to fill the intention stack with a fresh set of intentions.

The two agents as implemented by the two architectures, have identical knowledge bases, except for their programs and reward functions. That is, they believe the same actions are possible, with the same effects and associated probabilities. They both employ the exact same planner: $BestDoPO$. The graph in Figure 2 compares the performance of the two

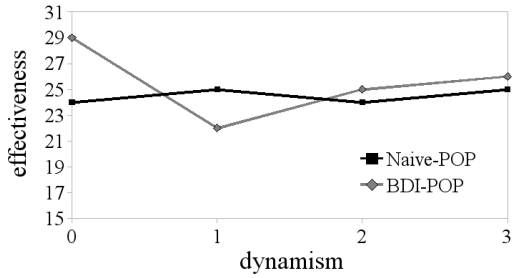

Figure 2: Performance of the two architectures.

architectures. Given the restrictions inherent in their respective architectures, each agent was roughly optimized to give them equal advantage.

Both architectures generate policies of horizon depth 3 ($h = 3$). In BDI-POP, we set $h^- = 1$—the search hori-

zon that $focus$ uses to determine program values. Throughout the experiments, the strategy for the time allowed to the agent was the same. There are 6 obstacles and initially 9 fires in each trial. We allow the agent to perform 3 actions each time before the obstacle positions change. The parameter for the number of obstacle changes per simulation cycle is the only parameter varied during experiments. Fourty trials per setting of this parameter were performed. *Effectiveness* is the total fire-points collected for the 40 trials. *Dynamism* is defined as 'number of obstacle changes per number of agent actions.

Conclusion

Compared to Naive-POP, the performance of BDI-POP in our experiments is not that impressive. This does not show that a BDI agent architecture should not be imbued with a POMDP planner; several enhancements to the simple BDI architecture used here are still possible: In particular, to build on this groundwork, we want to add the $reconsider$ predicate to deal with cases where intentions have become inappropriate to some degree, and utilizing partial/abstract plan structures.

Moreover, the relative sophistication of BDI-POP may not be applicable in very simple worlds such as FireEater. We would thus like to deploy our agents in a larger world, perhaps with more complicated tasks for the agent to perform. This will also give more scope for the variety of programs that would be applicable, and the real power of the BDI model could come into play.

What *has* been shown is that the proposed architecture is implementable; there is no obvious fundamental conflict in synthesizing our POMDP planner and the BDI model for agent control. The groundwork has thus been laid for the development of more sophisticated planning processes in the BDI-POP framework.

What remains unclear is how practical this approach might be in realistically complex domains. With probabilistic outcomes and events in the world, the policy searches blow up very quickly with depth. For complete and optimal policies, POMDP solvers can deal with just a modest number of easily enumerated states. Policy trees of a fixed depth (as generated by $BestDoPO$) are not complete policies and thus less costly to generate. Realistically though, due to belief states being extremely numerous and the equivalence problem for states in the situation calculus, $BestDoPO$ seems to be intractable if not undecidable. Furthermore, the situation calculus, in principle, provides a good deal of expressivity (including quantified reasoning), which brings its own computational complexity issues. For a hybrid BDI/POMDP architecture to scale up to a domain more meaningful than a microdomain, the integration of more 'common sense' reasoning techniques into the architecture may have benefits. And the latest advances in POMDP solvers (e.g., (Toussaint, Charlin, and Poupart 2008)) should be investigated for ideas to improve the efficiency of $BestDoPO$.

References

Bacchus, F.; Halpern, J.; and Levesque, H. 1999. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence* 1-2(111):171–208.

Boutilier, C.; Reiter, R.; Soutchanski, M.; and Thrun, S. 2000. Decision-theoretic, high-level agent programming in the situation calculus. In *Proc. AAAI-00*. AAAI Press. 355–362.

Bratman, M.; Israel, D.; and Pollack, M. 1988. Plans and resource-bounded practical reasoning. *Computational Intelligence* 4:349–355.

Bratman, M. 1987. *Intention, Plans, and Practical Reason*. Massachusetts/England: Harvard University Press.

Georgeff, M., and Ingrand, F. 1989. Decision-making in an embedded reasoning systems. In *Proc. IJCAI-89*. San Fransisco, CA: Morgan Kaufmann. 972–978.

Kaelbling, L.; Littman, M.; and Cassandra, A. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 1-2(101):99–134.

Levesque, H. J.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *J. of Log. Progr.* 31(1-3).

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and Thrun, S. 2005. Anytime dynamic A*: An anytime, replanning algorithm. In *Proc. Intl. Conf. on Automated Planning and Scheduling (ICAPS)*.

McCarthy, J. 1963. Situations, actions and causal laws. Technical report, Stanford University.

Pollack, M., and Ringuette, M. 1990. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proc. AAAI-90*. AAAI Press. 183–189.

Puterman, M. 1994. *Markov Decision Processes: Discrete Dynamic Programming*. New York, USA: Wiley.

Rao, A., and Georgeff, M. 1995. BDI agents: From theory to practice. In *Proc. ICMAS-95*. AAAI Press. 312–319.

Reiter, R. 2001. *Knowledge in Action*. MIT Press.

Rens, G.; Ferrein, A.; and Van der Poel, E. 2008. Extending DTGolog to deal with POMDPs. In *Proc. PRASA-08*. PRASA. 49–54.

Sardina, S.; De Silva, L.; and Padgham, L. 2006. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. AAMAS-06*. ACM Press. 1001–1008.

Toussaint, M.; Charlin, L.; and Poupart, P. 2008. Hierarchical POMDP controller optimization by likelihood maximization. In *Workshop on Advancements in POMDP Solvers, Tech. Report WS-08-01, AAAI-08*. AAAI Press. url:http://www.aaai.org/Library/Workshops/ws08-01.php.

Walczak, A.; Braubach, L.; Pokahr, A.; and Lambersdorf., W. 2007. Augmenting BDI agents with deliberative planning techniques. In *Proc. ProMAS-06*. Springer. 113–127.

Wooldridge, M. 1999. Intelligent agents. In Weiss, G., ed., *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Massachusetts/England: MIT Press. chapter 1.

Wooldridge, M. 2000. *Reasoning About Rational Agents*. Massachusetts/England: MIT Press.

Bibliography

[1] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, Massachusetts/England, 1998.

[2] F. Bacchus, J. Y. Halpern, and H. J. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence*, 111(1–2):171–208, 1999.

[3] D. Beck and G. Lakemeyer. Reinforcement learning for golog programs. In *Relational Approaches to Knowledge Representation and Learning, Workshop at the 32nd Annual Conference on Artificial Intelligence (KI-2009)*, 2009.

[4] M. Beetz, J. Hertzberg, M. Ghallab, and M. E. Pollack. *Advances in Plan-Based Control of Robotic Agents*. Springer Verlag, Berlin/Heidelberg, 2002.

[5] M. Beetz and A. Hofhauser. Plan-based control for autonomous soccer robots – preliminary report. In M. Beetz, J. Hertzberg, M. Ghallab, and M. E. Pollack, editors, *Advances in Plan-Based Control of Robotic Agents*, pages 21–35. Springer Verlag, Berlin/Heidelberg, 2002.

[6] G. A. Bekey. *Autonomous Robots: From Biological Inspiration to Implementation and Control*. MIT Press, Massachusetts/England, 2005.

[7] M. Boddy and T. Dean. Solving time-dependent planning problems. Technical report, Brown University, Providence, RI, 1989.

[8] R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artifical Intelligence*, 9(2–3):237–256, 1997.

[9] R. P. Bonasso. Introduction to part three. In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 193–194. AAAI Press / MIT Press, Menlo Park, CA / Massachusetts/England, 1998.

[10] B. Bonet and H. Geffner. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, 14(3):237–252, 2001.

[11] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[12] C. Boutilier, R. Reiter, and B Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the Seventeenth International. Joint Conference on Artificial Intelligence, IJCAI 2001*, pages 690–700, 2001.

[13] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pages 355–362. AAAI Press, Menlo Park, CA, 2000.

[14] R. J. Brachman and H. J. Levesque. *Knowledge representation and reasoning*. Morgan Kaufmann, San Fransisco, CA, 2nd edition, 2004.

[15] M. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Massachusetts/England, 1987.

[16] M. Bratman, D. Israel, and M. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

[17] D. Brutzman, T. Healey, D. Marco, and B. McGhee. The *Phoenix* autonomous underwater vehicle. In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 323–360. AAAI Press / MIT Press, Menlo Park, CA / Massachusetts/England, 1998.

[18] H. Burkhard, J. Bach, R. Berger, B. Brunswieck, and M Gollin. Mental models for robot control. In M. Beetz, J. Hertzberg, M. Ghallab, and M. E. Pollack, editors, *Advances in Plan-Based Control of Robotic Agents*, pages 71–88. Springer Verlag, Berlin/Heidelberg, 2002.

[19] A. Cassandra, L. Kaelbling, and M. Littman. Acting optimally in partially observable stochastic domains, CS-94-20. Technical report, Brown Universiteit Leuven, Department of Computer Science, Providence, Rhode Island, 1994.

[20] R. T. Clemen and T. Reilly. *Making hard decisions*. Duxbury, California, 2001.

[21] S. Commuri, J. S. Albus, and A. Barbera. Intelligent systems. In S. S. Ge and F. L. Lewis, editors, *Autonomous Mobile Robots: Sensing, Control, Decision Making and Applications*, pages 655–696. CRS Press, Boca Raton, FL, 2006.

[22] M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. Meyer. A programming language for cognitive agents: Goal directed 3APL. In *Proc. of 1st Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS03)*, 2003.

[23] G. de Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In *Logical Foundations for Cognitive Agents*, pages 86–102. Springer Verlag, 1998.

[24] L. de Silva and L. Padgham. Planning on demand in BDI systems. In *Proc. Intl. Conf. on Automated Planning and Scheduling (ICAPS-05)*. AAAI Press, 2005. Poster.

[25] D. Dennett. *The Intentional Stance*. MIT Press, Massachusetts/England, 1987.

[26] O. Despouys and F. F. Ingrand. Propice-plan: Toward a unified framework for planning and execution. In *ECP '99: Proceedings of the 5th European Conference on Planning*, pages 278–293, London, UK, 2000. Springer Verlag.

[27] G. Dudek and M. Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, New York, NY, 2000.

[28] A. Ferrein and G. Lakemeyer. Logic-based robot control in highly dynamic domains. *Journal of Robotics and Autonomous Systems, Special Issue on Semantic Knowledge in Robotics*, 2008.

[29] A. A. Ferrein. *Robot controllers for highly dynamic environments with real-time constraints*. PhD thesis, Knowledge-Based Systems Group, Rheinisch-Westfälischen Technischen Hochschule, Aachen, NW, 2007.

[30] A. Finzi and T. Lukasiewicz. Game-theoretic agent programming in Golog under partial observability. In *KI 2006: Advances in Artificial Intelligence*, volume 4314/2007, pages 113–127. Springer Verlag, Berlin / Heidelberg, 2007.

[31] R. J. Firby, P. N. Prokopowicz, and M. J. Swain. The animate agent architecture. In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 243–276. AAAI Press / MIT Press, Menlo Park, CA / Massachusetts/England, 1998.

[32] K. Fischer, J. P. Müller, and M. Pischel. A pragmatic BDI architecture. In M. Wooldridge, J. Müller, and M. Tambe, editors, *Intelligent Agents II: Agent Theories, Architectures, and Languages. Proc. of 2nd ATAL Workshop*, pages 203–218. Springer Verlag, Heidelberg/Berlin, 1996.

[33] S. P. Franklin. *Artificial Minds*. MIT Press, Massachusetts/England, 1995.

[34] C. Fritz and S. McIlraith. Compiling qualitative preferences into decision-theoretic GOLOG. In L. Morgenstern and M. Pagnucco, editors, *Proc. of IJCAI-05 Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC-05)*, pages 45–52. AAAI Press, 2005.

[35] A. Gabaldon and G. Lakemeyer. $\mathcal{ESP}$: A logic of only-knowing, noisy sensing and acting. In *Proc. of 22nd Natl. Conf. on Artificial Intelligence (AAAI-07)*, pages 974–979. AAAI Press, 2007.

[36] E. Gat. Three-layer architectures. In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 195–210. AAAI Press / MIT Press, Menlo Park, CA / Massachusetts/England, 1998.

[37] H. Geffner and J. Wainer. Modeling action, knowledge and control. In *Proc. of European Conference on Artificial Intelligence (ECAI-98)*, pages 532–536, 1998.

[38] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V: Agent Theories, Architectures, and Languages. Proc. of 5th ATAL Workshop*, pages 1–10. Springer Verlag, Heidelberg/Berlin, 1999.

[39] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning systems. In *Proc. of 6th Intl. Joint Conf. on Artificial Intelligence (IJCAI-89)*, pages 972–978. Morgan Kaufmann, San Fransisco, CA, 1989.

[40] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proc. of 6th Natl. Conf. on AI (AAAI-87)*, pages 677–682, Seattle, WA, 1987. AAAI Press.

[41] E. B. Goldstein. *Cognitive Psychology: Connecting Minds, Research, and Everyday Experience*. Thomson Wadsworth, Belmont, CA, 2005.

[42] H. Grosskreutz. *Towards more realistic logic-based robot controllers in the Golog framework*. PhD thesis, Knowledge-Based Systems Group, Rheinisch-Westfälischen Technischen Hochschule, 2002.

[43] K. V. Hendriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming with declarative goals. In *Intelligent Agents VII: Agent Theories, Architectures, and Languages. Proc. of 7th ATAL Workshop*, pages 228–243. Springer Verlag, Heidelberg/Berlin, 2001.

[44] M. J. Huber. JAM: a BDI-theoretic mobile agent architecture. In *AGENTS-99: Proc. of 3rd Annual Conf. on Autonomous Agents*, pages 236–243, New York, NY, 1999. ACM Press.

[45] J. F. Hübner, R. H. Bordini, and M. Wooldridge. Programming declarative goals using plan patterns. In *Declarative Agent Languages and Technologies IV (DALT-06), LNAI, Vol. 4327*, pages 123–140. Springer Verlag, Berlin / Heidelberg, 2006.

[46] F. Ingrand, R. Chatila, R. Alami, and F. Rober. PRS: A high level supervision and control language for autonomous mobile robots. In *Proc. of IEEE International Conference on Robotics and Automation (ICRA-96)*, 1996.

[47] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert, Knowledge-Based Diagnosis in Process Engineering*, 7(6):34–44, 1992.

[48] L. Kaelbling, M. Littman, and A. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998.

[49] L. Karlsson and T. Schiavinotto. Progressive planning for mobile robots – a progress report. In M. Beetz, J. Hertzberg, M. Ghallab, and M. E. Pollack, editors, *Advances in Plan-Based Control of Robotic Agents*, pages 106–122. Springer Verlag, Berlin/Heidelberg, 2002.

[50] S. M. Khan and Y. Lespérance. A logical account of prioritized goals and their dynamics. In G. Lakemeyer, L. Morgenstern, and M-A. Williams, editors, *Proc. of 9th Intl. Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense 2009)*, pages 85–90, University of Technology, Sydney, 2009. UTSe Press.

[51] I. Kim, H. Shin, and J. Choi. A plan-based control architecture for intelligent robotic agents. In *Proc. of 1st KES Symposium (KES-AMSTA-07)*, pages 559–567, Heidelberg/Berlin, 2007. Springer Verlag.

[52] D. Kinny and M. Georgeff. Commitment and effectiveness of situated agents. In *Proc. of 12th Intl. Joint Conf. on Artificial Intelligence (IJCAI-91)*, pages 82–88, 1991.

[53] D. Kinny and M. Georgeff. Experiments in optimal sensing for situated agents. In *Proc. of the 2nd Pacific Rim Intl. Conf. on Artificial Intelligence (PRICAI-92)*, 1992.

[54] K. Kolonige and K. Myers. The Saphira architecture for autonomous mobile robots. In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 211–242. AAAI Press / MIT Press, Menlo Park, CA / Massachusetts/England, 1998.

[55] D. Kortenkamp, R. P. Bonasso, and R. Murphy. Introduction: Mobile robots—a proving ground for AI. In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 3–18. AAAI Press / MIT Press, Menlo Park, CA / Massachusetts/England, 1998.

[56] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *Proc. of AAAI-02*, pages 447–454. AAAI Press/The MIT Press, 2002.

[57] J. E. Laird. Extending the Soar cognitive architecture. In *Artificial General Intelligence 2008*, pages 224–235. IOS Press, Amsterdam, The Netherlands, 2008.

[58] J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33:l–64, 1987.

[59] G. Lakemeyer and H. J. Levesque. $\mathcal{AOL}$: A logic of acting, sensing, knowing, and only knowing. In *Proc. Principles of Knowledge Representation and Reasoning (KR-98)*, pages 316–327, 1998.

[60] J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *Proc. Conf. on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS-94)*, pages 842–849, 1994.

[61] H. J. Levesque and G. Lakemeyer. Situations, si! Situation terms no! In *Proc. Principles of Knowledge Representation and Reasoning (KR-04)*, pages 516–526. AAAI Press, 2004.

[62] H. J. Levesque and G. Lakemeyer. Cognitive Robotics. In B. Porter F. van Harmelen, V. Lifshitz, editor, *The Handbook of Knowledge Representation*, pages 869–886. Elsevier Science, 2007.

[63] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–84, 1997.

[64] V. Lifschitz. On the semantics of Strips. In *In Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, San Mateo, CA, 1987. Morgan Kaufmann.

[65] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *Proc. Intl. Conf. on Automated Planning and Scheduling (ICAPS)*, 2005.

[66] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[67] F. R. Meneguzzi, A. F. Zorzo, and M. da Costa Móra. Propositional planning in BDI agents. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 58–63, New York, NY, USA, 2004. ACM.

[68] E. Messina and S. Balakirsky. Knowledge representation and decision making for mobile robots. In S. S. Ge and F. L. Lewis, editors, *Autonomous Mobile Robots: Sensing, Control, Decision Making and Applications*, pages 465–500. CRS Press, Boca Raton, FL, 2006.

[69] A. M. Meystel and J. S. Albus. *Intelligent Systems: Architectures, Design, and Control*. John Wiley & Sons, Inc., New York, 2002.

[70] J. P. Müller. The right agent (architecture) to do the right thing. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V: Agent Theories, Architectures, and Languages. Proc. of 5th ATAL Workshop*, pages 211–225. Springer Verlag, Heidelberg/Berlin, 1999.

[71] R. R. Murphy. *Introduction to AI Robotics*. MIT Press, Massachusetts/England, 2000.

[72] K. L. Myers. A procedural knowledge approach to task-level control. In B. Drabble, editor, *Proc. of 3rd AI Planning Systems Conf.*, pages 158–165. AAAI Press, 1996.

[73] N. J. Nilsson. A mobile automaton: An application of artificial intelligence techniques. Technical report, Stanford Research Institute, Menlo Park, CA, 1969.

[74] N. Oliver and E. Horvitz. Selective perception policies for guiding sensing and computation in multimodal systems: A comparative analysis. In *Proceedings of the Fifth International Conference on Multimodal Interaction*. ACM Press, Vancouver, 2003.

[75] M. Paolucci, O. Shehory, K. P. Sycara, D. Kalp, and A. Pannu. A planning component for retsina agents. In *Intelligent Agents VI: Agent Theories, Architectures, and Languages. Proc. of 6th ATAL Workshop*, pages 147–161, London, UK, 2000. Springer Verlag.

[76] J. Pineau. *Tractable Planning Under Uncertainty: Exploiting Structure*. PhD thesis, Robotics Institute, Carnegie Mellon University, 2004.

[77] M. Pollack and M. Ringuette. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proc. of AAAI-90*, pages 183–189. AAAI Press, 1990.

[78] J. L. Pollock. The logical foundations of goal-regression planning in autonomous agents. *Artificial Intelligence*, 106:267–335, 1998.

[79] J. L. Pollock. OSCAR: An architecture for generally intelligent agents. In *Artificial General Intelligence 2008*, pages 275–286. IOS Press, Amsterdam, The Netherlands, 2008.

[80] D. Poole. Decision theory, the situation calculus and conditional plans. *Linköping Electronic Articles in Computer and Information Science*, 8(3), 1998.

[81] A. Rao and M. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proc. Principles of Knowledge Representation and Reasoning (KR-91)*, pages 473–484, San Mateo, CA, 1991. Morgan Kaufmann.

[82] A. Rao and M. Georgeff. BDI agents: From theory to practice. In *Proc. of ICMAS-95*, pages 312–319. AAAI Press, 1995.

[83] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proc. of 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96)*, pages 42–55, Berlin/Heidelberg, 1996. Springer Verlaag.

[84] R. Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT Press, Massachusetts/England, 2001.

[85] G. Rens, A. Ferrein, and E. van der Poel. Extending DTGolog to deal with POMDPs. In F. Nicolls, editor, *Proc. of 19th Annual Symposium of the Pattern Recognition Association of South Africa (PRASA-08)*, pages 49–54, Cape Town, South Africa, 2008. UCT Press. url: http://hdl.handle.net/10204/2972.

[86] G. Rens, A. Ferrein, and E. van der Poel. A BDI agent architecture for a POMDP planner. In G. Lakemeyer, L. Morgenstern, and M-A. Williams, editors, *Proc. of 9th Intl. Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense 2009)*, pages 109–114, University of Technology, Sydney, 2009. UTSe Press.

[87] G. B. Rens and I. J. Varzinczak. Introducing a logic for real-world agents with degrees of belief. In F. Nicolls, editor, *Proc. of 20th Annual Symposium of the Pattern Recognition Association of South Africa (PRASA-09)*, page 146, 2009. Poster abstract.

[88] S. J. Russell and P. Norvig. *Artificial intelligence: A modern approach*. Prentice Hall, New Jersey, 2nd edition, 2003.

[89] S. Russell S. Zilberstein. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1–2):181–213, 1996.

[90] S. Sardina, G. de Giacomo, Y. Lespérance, and H. J. Levesque. On the semantics of deliberation in IndiGolog: from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41:259–299, 2004.

[91] S. Sardina, L. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. of AAMAS-06*, pages 1001–1008. ACM Press, 2006.

[92] S. Sardina and Y. Lesprance. Golog speaks the BDI language (to appear). In *Programming Multi-Agent Systems, 7th International Workshop (ProMAS-09). Revised Invited and Selected Papers, LNCS, Vol. 5919*. Springer Verlag, 2009. url: http://www.cse.yorku.ca/ lesperan/papers/PROMAS09LNCS.pdf.

[93] L. J. Savage. *The Foundations of Statistics*. Dover Publications, Dover, NY, 2nd edition, 1972.

[94] F. Schönherr and J. Hertzberg. The DD&P robot control architecture. In M. Beetz, J. Hertzberg, M. Ghallab, and M. E. Pollack, editors, *Advances in Plan-Based Control of Robotic Agents*, pages 249–269. Springer Verlag, 2002.

[95] M. Schut and M. Wooldridge. Intention reconsideration in complex environments. In *Proc. of the 4th Intl. Conf. on Autonomous Agents (AGENTS-00)*, pages 209–216, New York, NY, USA, 2000. ACM.

[96] M. Schut and M. Wooldridge. The control of reasoning in resource-bounded agents. *The Knowledge Engineering Review*, 16(3):215–240, 2001.

[97] M. Schut and M. Wooldridge. Principles of intention reconsideration. In *Agents 2001: Proc. of 5th Intl. Conf. on Autonomous Agents*, pages 340–347, New York, NY, 2001. ACM Press.

[98] M. Schut, M. Wooldridge, and S. Parsons. The theory and practice of intention reconsideration. *Experimental and Theoretical Artificial Intelligence*, 16(4):261–293, 2004.

[99] S. Sohrabi, J. A. Baier, and S. A. McIlraith. Htn planning with preferences. In G. Lakemeyer, L. Morgenstern, and M-A. Williams, editors, *Proc. of 9th Intl. Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense 2009)*, pages 115–122, University of Technology, Sydney, 2009. UTSe Press.

[100] M. Soutchanski. *High-Level Robot Programming in Dynamic and Incompletely Known Environments*. PhD thesis, Graduate Department of Computer Science, University of Toronto, 2003.

[101] R. J. Sternberg. *Cognitive Psychology*. Thomson Wadsworth, Belmont, CA, 3rd edition, 2003.

[102] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[103] C. Thorne. The BDI model of agency and BDI logics. Technical report, L.O.A. - C.N.R., Trento, May 2005.

[104] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, Massachusetts/England, 2005.

[105] M. Toussaint, L. Charlin, and P. Poupart. Hierarchical POMDP controller optimization by likelihood maximization. In *Workshop on Advancements in POMDP Solvers, Tech. Report WS-08-01, AAAI-08*. AAAI Press, 2008. url: http://www.aaai.org/Library/Workshops/ws08-01.php.

[106] I. J. Varzinczak. How do I revise my agent's action theory? In G. Lakemeyer, L. Morgenstern, and M-A. Williams, editors, *Proc. of 9th Intl. Symposium on Logical Formal-*

izations of Commonsense Reasoning (Commonsense 2009), pages 129–134, University of Technology, Sydney, 2009. UTSe Press.

[107] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, 3rd edition, 1953.

[108] A. Walczak, L. Braubach, A. Pokahr, and W. Lamersdorf. Augmenting BDI agents with deliberative planning techniques. In R. H. Bordini, M. Dastani, J. Dix, and A. Seghrouchni, editors, *Proc. of 4th Intl. Workshop of Programming Multi-Agent Systems (ProMAS-06)*, pages 113–127, Heidelberg/Berlin, 2007. Springer Verlag.

[109] M. Winikoff. An AgentSpeak meta-interpreter and its applications. In *Programming Multi-Agent Systems (ProMAS-05), LNAI Vol. 3862*, pages 123–138. Springer Verlag, Berlin / Heidelberg, 2006.

[110] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proc. of KR-02*, pages 407–481, 2002.

[111] M. Wooldridge. Intelligent agents. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter 1. MIT Press, Massachusetts/England, 1999.

[112] M. Wooldridge and S. Parsons. Intention reconsideration reconsidered. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V: Agent Theories, Architectures, and Languages. Proc. of 5th ATAL Workshop*, pages 63–80. Springer Verlag, Heidelberg/Berlin, 1999.

[113] M. J. Wooldridge. *Reasoning about Rational Agents*. MIT Press, Massachusetts/England, 2000.

[114] M. J. Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, Chichester, England, 2002.

[115] M. J. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: A survey. In *Intelligent Agents, LNCS, Vol. 890*, pages 1–39. Springer Verlag, Berlin/Heidelberg, 1995.

[116] S. Zilberstein, R. Washington, D. S. Bernstein, and A. Mouaddib. Decision-theoretic control of planetary rovers. In M. Beetz, J. Hertzberg, M. Ghallab, and M. E. Pollack, editors, *Advances in Plan-Based Control of Robotic Agents*, pages 270–289. Springer Verlag, Berlin/Heidelberg, 2002.