HIGH-LEVEL ROBOT PROGRAMMING IN DYNAMIC AND INCOMPLETELY KNOWN ENVIRONMENTS.

by

Mikhail Soutchanski

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

High-Level Robot Programming in Dynamic and Incompletely Known Environments.

Mikhail Soutchanski

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2003

This thesis advocates the usefulness and practicality of a logic-based approach to AI and in particular to high-level control of mobile robots. The contribution of the research work reported here is twofold: 1) the development of theoretical frameworks that account for uncertainty and unmodeled dynamics in an environment where an acting agent has to achieve certain goals and 2) the implementation of the developed ideas on a mobile robot.

We have elaborated the approach to designing efficient and reliable controllers in Golog following two different perspectives on the environment where the control program is supposed to operate.

According to one perspective, investigated in Chapter 4, the agent has a logical model of the world, but there is no probabilistic information about the environment where the agent is planning to act, and the agent is not capable or has no time for acquiring probabilities of different effects of its actions. In this case, the uncertainty and dynamics of the environment can be accounted only by observing the real outcomes of actions executed by the agent, by determining possible discrepancies between the observed outcomes and the effects expected according to the logical model of the world and then by recovering, if necessary, from the relevant discrepancies. To recover the agent computes on-line an appropriate correction of the program that is being executed. A general framework for execution monitoring of Golog programs provides the aforementioned functionalities and generalizes those previously known approaches to execution monitoring that have been formulated only for cases when the agent

is given a linearly or partially ordered *sequence* of actions, but not an arbitrary *program*.

According to the second perspective, investigated in Chapter 5, we can model actions of the agent as stochastic actions and characterize them by a finite set of probabilities: whenever the agent does a stochastic action, it may lead to a finite number of possible outcomes. Two major innovations in this research direction are the development of a decision-theoretic Golog (DT-Golog) interpreter, that deals with programs that include stochastic actions, and the development of the situation calculus representation of MDPs. In addition to this *off-line* DT-Golog interpreter, in Chapter 6 we develop an *on-line* DT-Golog interpreter that combines planning with the execution of policies. This new on-line architecture allows one to compute an optimal policy (optimal with respect to a given Golog program and a current model of the world) from an initial segment of a Golog program, execute the computed policy on-line and then proceed to computing and executing policies for the remaining segments of the program. The specification and implementation of the on-line interpreter requires a new approach to the representation of sensing actions in the situation calculus. A formal study of this approach is undertaken in Chapter 3. We also describe implementations of our frameworks; these were successfully tested in a real office environment on a mobile robot B21.

Acknowledgements

The long period of writing this Ph.D. thesis has given me the opportunity to reflect on all of the people who influenced me, taught me, and supported me in diverse ways throughout the project. Foremost among these is my mentor, Ray Reiter, who gave me the freedom to pursue my ideas and interests, and also taught me the principles of good scientific research in AI. His wise guidance and vision, and his passion for fundamental research that leads to long-term progress, were important factors that greatly inspired the work reported in this thesis. Ray's enthusiasm for logic-based AI and for the new research area of cognitive robotics was infectious and influential. His untimely death on September 16, 2002, was a major loss. This thesis is dedicated to his memory. My most sincere thanks to Hector Levesque, who took over the responsibilities as supervisor after Ray's death and inspired my final push to graduation. Hector played the role of informal co-supervisor throughout my Ph.D. career; in the last stage of this work, he was particularly helpful and supportive. His astute critical comments and suggestions on thesis drafts were always well targeted. The other members of the committee, Craig Boutilier, Yves Lesperance and John Mylopolous, have also contributed much to the shaping of this thesis with their comments and advice. Craig Boutilier was generous in sharing expertise in methods of dealing with uncertainty in AI, and he had an important influence on the direction of the research. John Mylopolous contributed both by making incisive comments and by providing expertise in requirements engineering. Yves Lesperance read several drafts of this thesis and provided many comments and insightful suggestions; these improved the work's presentation and helped to clarify subtle issues. My special thanks to Fiora Pirri, my external examiner, for making several useful suggestions for improving this thesis. In addition, I am in debt to the researchers who contributed to the development of BeeSoft, the robotics software that was used extensively in my research implementation. In particular, I would like to thank Dieter Fox, Dirk Haehnel and Sebastian Thrun: without them, it would have been impossible to run my programs on a real mobile robot. Many thanks to the researchers who shared their insights or provided useful comments at different stages of this thesis work: most importantly, Joachim Hertzberg, Erik Sandewall, Murray Shanahan, Michael Thielscher, and Sebastian Thrun. The intellectual debt of this thesis to John McCarthy is unquestionable.

Over the years, I have enjoyed the continuous encouragement and support of members of the Cognitive Robotics group in Toronto: in particular, Alfredo Gabaldon, Yilan Gu, Hojjat Ghaderi, Sam Kaufman, Iluju Kiringa, Yongmei Liu, Sheila McIlraith, Ron Petrick, and Sebastian Sardina. I would also like to thank external collaborators: Giuseppe De Giacomo, Gerhard Lakemeyer, Fangzhen Lin and Maurice Pagnucco. Thanks particularly to Giuseppe De Giacomo for lively debates and for sharing insights and expertise in logic programming, and to Gerhard Lakemeyer for encouragement. I owe special thanks to Iluju Kiringa, my friend and office mate, for many interesting conversations. Thanks also to other former office mates - Luis Dissett, Tomo Yamakami, and other members of the local Computer Science community (Marsha Chechik, Sherif Ghali, Antonina Kolokolova, David Modjeska, Natalia Modjeska, Lucia Moura, Daniel Panario, Marcus Santos and others) for keeping my spirits high. A great debt of gratitude goes to my long-term friends Raisa and Vladimir Ivashko, Irina and Mikhail Vyalyi, Alla and Andrei Shakhovskoi, Nadezhda and Vladimir Makarov, Tatyana and Victor Lyutyi, Elena and Vladimir Aristov, Gulshat and Evgeni Nadezhdin, and especially to Vladimir Kuznecov and Yuri Derlyuk for their understanding of my academic pursuits and unconditional friendship.

The following organizations provided valuable financial assistance: the Department of Computer Science of the University of Toronto, NSERC (the National Sciences and Engineering Research Council of Canada), and IRIS-Precarn (the Canadian Institute for Robotics and Intelligent Systems).

Finally, this dissertation could not have been completed without the love, understanding and support of Irina Dyakova and my mother, Sabina Soutchanski.

Contents

Chapter 1

Introduction

1.1 Motivation

This thesis advocates the usefulness and practicality of a logic-based approach to AI and in particular to high-level control of mobile robots. The contribution of the research work reported here is twofold: 1) the development of theoretical frameworks that account for uncertainty and unmodeled dynamics in an environment where an acting agent has to achieve certain goals and 2) the implementation of the developed ideas on a mobile robot.

There are several advantages of a logic-based approach to AI [McCarthy, 1990, Reiter, 2001a]. First, knowledge about the world can be represented in a logical theory independent of how this knowledge will be used to solve practical tasks (declarativeness). Second, it is possible to incorporate high-level cognitive skills such as planning, reasoning about other agents, and commonsense reasoning into a logic-based agent (cognitive adequacy). Third, an accessible logic-based model of the world can be used to compute appropriate responses required to achieve reliability during the execution of plans in a dynamic and uncertain world (robustness). Fourth, because essential specifications are expressed in logic, in cases when our implementation is sound with respect to the logical specification, we can verify formally whether our agent is capable of achieving its goals safely (verification). Fifth, if knowledge about the world and goals of an agent are expressed in a logical language, then they can be relatively easy to modify (elaboration tolerance).

1.2 Methodology

Intelligent agents operating in realistic environments are confronted with an uncertain, dynamic world and given only partial information about its initial state. To operate successfully and achieve their goals, those agents must be provided with models that account for the complexity of real environments. For this reason, the task of adequate modeling of the world is the necessary preliminary step towards designing complex controllers for intelligent agents. In the past few years, it has been demonstrated that predicate logic-based frameworks provide all the required expressive power and flexibility for constructing adequate models. More specifically, it has been demonstrated that many sophisticated models can be expressed in the situation calculus (a popular knowledge representation framework that is entirely formulated in the classical predicate logic). Recently, the situation calculus has been extensively studied and modified, and the version formulated in [Levesque *et al.*, 1998, Reiter, 2001a] is a well-developed approach to axiomatizing effects of primitive actions on the world. In addition, the Cognitive Robotics group at the University of Toronto has developed the high-level logic-based programming language Golog [Levesque *et al.*, 1997, Levesque and Reiter, 1998]. Golog has all standard programming constructs and several non-deterministic operators; it can be used to compose complex controllers from primitive actions specified in the situation calculus. In particular, the planning task can be done using search constrained by the given nondeterministic program.

These recent developments not only introduced a very general, semantically clear approach to modeling dynamic systems in classical predicate logic, but also brought about techniques for designing computationally efficient high-level controllers: by varying the degree of nondeterminism in a Golog program and by using operators that bound the scope of search, the programmer can influence the efficiency of search for an executable control sequence. From the perspective of my research, Golog is an ideal language because at each step of a computation, the logical statements about the current logical model can be easily evaluated; in other words, the logical model of the world is easily accessible.

1.3 Outline of the Thesis

The main objective of my PhD research is accounting for uncertainty and unmodeled dynamics in an environment. To achieve this I have developed several innovative proposals. I have elaborated some approaches to designing efficient and reliable controllers in Golog following two different perspectives on the environment where the control program is supposed to operate.

1.3.1 A Deterministic Perspective.

According to one perspective, there is no probabilistic information about the environment where the agent is planning to act, and the agent is not capable or has no time for acquiring probabilities of different effects of its actions (this includes the case when estimation of those probabilities is impractical due to the nature of application domain). This setting is common in cases when the robot has to explore a previously unfamiliar environment only once or a few times: due to scarcity of interaction experiences, the probabilistic model cannot be estimated in advance. In this case, the uncertainty and dynamics of the environment can be dealt with only by observing the real outcomes of actions executed by the agent, by determining possible discrepancies between the observed outcomes and the effects expected according to the logical model of the world and then by recovering, if necessary, from the relevant discrepancies: to recover the agent computes on-line an appropriate correction of the program that is being executed. A general framework for execution monitoring of Golog programs provides the aforementioned functionalities and generalizes those previously known approaches to execution monitoring that have been formulated only for cases when the agent is given a linearly or partially ordered *sequence* of actions, but not an arbitrary *program*. Chapter 4 considers two general recovery mechanisms that can be employed by the execution monitor: a planning procedure and backtracking, respectively. The planning procedure computes on-line a short corrective sequence of actions such that after doing those actions, the agent can resume executing the remaining part of the program. Backtracking to previous choice points in the non-deterministic Golog program gives an agent a chance to choose an alternative execution branch in cases when the further execution of actions remaining in the current branch is no longer possible (a post-condition of the program will not be satisfied) or desirable (there are other execution branches with higher utility). For example, this happens if the last action was executed overly late and no matter how fast the agent will execute actions remaining in the branch, it will miss a deadline at the end. In the general case, when the agent has executed actions after choosing a particular branch of a non-deterministic Golog program, the generalized recovery mechanism may use both planning and backtracking procedures if it is not possible or desirable to continue the execution of the current branch: the planning procedure is responsible in this case for computing a plan that leads back to a program state where an alternative branch can be chosen.

Both recovery mechanisms have been successfully implemented. In particular, the execution monitoring with backtracking has been implemented on the mobile robot Golem (B21 manufactured by Real World Interface, Inc). This implementation has demonstrated that given a non-deterministic Golog program with actions that have an explicit temporal argument, the monitor computes on-line an alternative schedule of deliveries in a real office environment, when the robot encounters unmodeled obstacles that unexpectedly delay its travel between offices. The low-level software was initially developed at the University of Bonn and Carnegie-Mellon University to control Rhino, another B21 robot, which is famous for its use in museum guide applications; see [Burgard *et al.*, 1998, Burgard *et al.*, 1999, Thrun *et al.*, 1999] for details.

I expect that thanks to its generality, the execution monitoring framework can find applicability in domains outside robotics (e.g., Semantic Web, software engineering). The reliability provided by this framework to high-level logic-based programs in dynamic and uncertain environments can be achieved without extra effort for a Golog programmer. The logical model of the domain, once it is constructed, is the only component required by the monitor to function properly and make sure that the program runs robustly.

1.3.2 A Probabilistic Decision-Theoretic Perspective.

According to the second perspective, we can model actions of the agent as stochastic actions and characterize them by a finite set of probabilities: whenever the agent does a stochastic action, it may lead to a finite number of possible outcomes (states). Each outcome (state) is associated with a certain reward value, and each sequence of actions is characterized by the expected utility representing rewards accumulated over all possible sequences of outcomes. In this case, the interaction of the agent with the environment can be described by a Markov Decision Process (MDP), and the agent has to solve the corresponding decision-theoretic planning (DTP) problem: compute an optimal policy. Here, a policy is a conditional Golog program with test conditions serving to identify a real outcome of stochastic actions; an optimal policy is one that yields the maximum expected utility.

In realistic domains, MDPs have certain structure and it is natural to exploit this structure to solve efficiently the corresponding DTP problems even if the state space of an MDP is very large. More specifically, a Golog programmer can design a Golog program that provides natural constraints on the search for an optimal policy. Two major innovative steps in this research endeavor were the development of a decision-theoretic Golog (DT-Golog) interpreter, that deals with programs that include stochastic actions, and the development of a situation calculus representation of MDPs. By using the domain specific procedural knowledge expressed in the Golog program, an off-line DT-Golog interpreter computes a policy optimal with respect to constraints in the program given a reward function and the situation calculus-based representation of an MDP. This gives to the programmer unique flexibility in designing complex controllers: both the opportunity to design efficient controllers and the simplicity of leaving the solution of the problem of what constitutes an optimal policy to the off-line interpreter. In Chapter 5, we provide arguments that the proposed approach allows for the seamless integration of programming and planning. In particular, the chapter outlines a robotics implementation and examples that demonstrate the importance of having a logical model of the world: using a situation calculus based representation of an MDP and a Golog program that provides natural constraints, the DT-Golog interpreter is able to compute an optimal policy even in the case when the state space of the associated MDP has several billion states.

The interpreter specified in Chapter 5 is an *off-line* interpreter: it computes an optimal policy and only after that the policy is executed on-line. In Chapter 6, we discuss an *on-line* DT-Golog interpreter that combines planning with the execution of policies. This on-line architecture allows one to compute an optimal policy (optimal with respect to a given Golog program and a current model of the world) from an initial segment of a Golog program, execute the computed policy on-line and then proceed to computing and executing policies for the remaining segments of the program. The specification and implementation of the on-line interpreter requires a new approach to the representation of sensing actions in the situation calculus. A formal study of this approach is undertaken in Chapter 3. The on-line DT-Golog interpreter was also successfully tested in the real office environment on the same robot. The implementation proved the efficiency of this new on-line architecture.

Chapter 2

Background

This chapter provides background material that we use in all subsequent chapters of this thesis.

2.1 The Situation Calculus with Time and Knowledge

In this section, we collect definitions and concepts related to representation and reasoning about actions. We choose the situation calculus as a logical representation language. Specifically, we introduce three dialects of the situation calculus: the basic version, the temporal situation calculus and the situation calculus with an explicit epistemic fluent.

2.1.1 The Basic Situation Calculus

The situation calculus [McCarthy, 1963] is a first-order language for axiomatizing dynamic worlds. In 1990s, it has been considerably extended beyond the initial proposal to include concurrency, continuous time, nondeterminism, etc, but in all cases, its basic ingredients consist of *actions*, *situations* and *fluents*.

Let $\mathcal{L}$ be a sorted first-order language with three disjoint sorts for actions, situations and objects. [1] The vocabulary of $\mathcal{L}$ consists of symbols described below. We will use Greek letters ϕ, ψ and ξ to denote well-formed formulas. Variables of sort *action* will be denoted $a, \alpha, a', a_1, \alpha_2 \ldots$; variables of sort *situation* will be denoted $s, s', s_1, s_2, \sigma, \sigma_1, \sigma_2, \ldots$; *object* variables will be denoted by x, y and other letters and English words.

[1] To simplify definitions the sort object includes agents, physical entities, and locations.

Actions

There are finitely many different function symbols of sort *action*; arguments of such symbols may be only of sort *object*. We will name these function symbols primitive *actions*. Informally, we understand primitive actions in this subsection as deterministic physical ones: e.g., as manipulations with physical objects: put, take, grasp (stochastic actions will be considered in Chapter 5).

Terms of sort *action* also will be denoted by lower-case letters a and α with primes and subscripts, and arguments will be shown explicitly whenever we want to stress the difference between variables and terms of sort *action*.

Situations

Function symbols of sort *situation*: there are just two of these — the constant S_0, and the binary function symbol do which takes arguments of sort *action* and *situation*. Upper-case letters S with primes and subscripts denote ground *situation* terms. The constant S_0 denotes an initial situation, $do(a, s)$ denotes a situation resulting from the execution of action a in situation s. We understand *situation* is a first-order term denoting a sequence of actions: $do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. The special constant S_0 denotes the empty action sequence; so S_0 is like LISP's ( ) or Prolog's [ ]. Therefore, the situation term

$$do(putOn(Book, Desk), do(grasp(Book),$$
$$do(goFromTo(Hall, Room), S_0)))$$

denotes the sequence of actions: $goFromTo(loc, loc')$, $grasp(Book)$, $putOn(Book, Desk)$ Notice that the action sequence is obtained from a situation term by reading the term from right to left. For this reason, it is convenient to use the following suggestive notation: $do([a_1, \ldots, a_n], s)$, where $a_1, \ldots, a_n$ are terms of sort *action*:

$$do([\,], s) = s,$$

$$do([a_1, \ldots, a_n], s) = do(a_n, do([a_1, \ldots, a_{n-1}], s)) \quad n = 1, 2, \ldots$$

Thus, $do([a_1, \ldots, a_n], s)$ is a compact notation for the situation term $do(a_n, do(a_{n-1}, \ldots do(a_1, s) \ldots))$ which denotes that situation resulting from performing the action a_1, followed by $a_2, \ldots$, followed by a_n, beginning in situation s.

Fluents

Relations or functions whose values may vary from situation to situation are called *fluents*. They are denoted by predicate or function symbols; each symbol takes, among its arguments, exactly one of sort *situation*. For notational convenience, we assume that the *situation* argument is the last argument of fluents. There are finitely many fluents; these are domain specific predicate or function symbols. For example, $closeTo(x, y, s)$ might be a relational fluent, meaning that x will be close to y in the result of performing the action sequence s; $pos(x, s)$ might be a functional fluent, denoting x's position in the result of performing the action sequence s.

Following [Reiter, 2001a], the language of the situation calculus includes a distinguished equality symbol $=$and two other predicate constants:

1. A distinguished binary predicate symbol $Poss$ taking arguments of sort *action* and *situation*, respectively; $Poss(a, s)$ means action a is possible in situation s.
2. A distinguished binary predicate symbol $\sqsubset$ taking arguments of sort *situation*. We will use $s \sqsubseteq s'$ as shorthand for $s \sqsubset s' \vee s = s'$.

There are also finitely many other domain specific function symbols of sort *object* for each arity none of which take an argument of sort *situation* or an argument of sort *action*. The language may also include finitely many domain specific predicate symbols which may have an argument of sort *action*, but have no arguments of sort *situation*. We use ordinary logical connectives and punctuation.

Basic Action Theories

Let Σ be the set of the foundational axioms:

$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2, \tag{2.1}$$

$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s), \tag{2.2}$$

$$\neg s \sqsubset S_0, \tag{2.3}$$

$$s \sqsubset do(a, s') \equiv s \sqsubseteq s'. \tag{2.4}$$

The axiom (2.2) is a second order induction axiom saying that the sort *situation* is the smallest set containing S_0 that is closed under the application of *do* to an *action* and *situation*. The axiom (2.1) is a unique names axiom for situations that together with (2.2) imply that if two situations are the same, then each of them is the result of the same sequence of actions applied in S_0 (informally, different sequences of actions forking from S_0 cannot join in the same situation). The axiom (2.3) means that S_0 has no predecessors, and (2.4) asserts that a situation s is a predecessor of (or equal to) other situation that results from doing an action a in a situation s' if and only if s is a predecessor of s' or s is equal to s'. These axioms are domain independent and must be included in any domain specific axiomatization of an application.

In addition to foundational axioms, it is convenient to characterize only those situations $do([a_1, \ldots, a_n], s)$ in which all actions are actually possible to execute one after the other. These situations are called *reachable* and they are defined by an abbreviation:

$$reachable(s) \stackrel{def}{=} (\forall a, s').\ do(a, s') \sqsubseteq s \supset Poss(a, s'). \tag{2.5}$$

A domain theory is axiomatized in the situation calculus with the following four classes of axioms ([Pirri and Reiter, 1999, Reiter, 2001a]):

Unique names axioms for actions $\mathcal{D}_{una}$. These axioms state that the actions of the domain are pairwise unequal.

Initial theory of the world $\mathcal{D}_{S_0}$. This is a set of first-order sentences whose only situation term is S_0. This set of axioms specifies what must hold before any actions have been "executed". Thus, no fluent of a formula of $\mathcal{D}_{S_0}$ mentions a variable of sort *situation* or the function symbol do; $\mathcal{D}_{S_0}$ characterize only the initial *situation*.

Action precondition axioms $\mathcal{D}_{ap}$. There is one axiom for each primitive action $A(\vec{x})$:

$$Poss(A(\vec{x}), s) \ \equiv\ \Pi_A(\vec{x}, s),$$

where $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$. Each axiom characterizes the preconditions of action $A(\vec{x})$: the action can be executed in situation s only if the formula $\Pi_A(\vec{x}, s)$ holds in s. Each precondition axiom has only one occurrence of the predicate constant *Poss*; the formula $\Pi_A(\vec{x}, s)$ has neither occurrences of predicate constants $Poss$ and $\sqsubset$ nor occurrences of the term *do*, it mentions only one situation variable s and it does not include quantifiers over this situation variable. Any formula that satisfy all these syntactic requirements is called *uniform in* s. Uniform formulas are typical in domain axiomatizations. Note that because the right hand side of the precondition axiom is an uniform formula, an action $A(\vec{x})$ is possible to execute s only if some conditions apply exactly in s (not in situations that precede s or in situations for which s is a predecessor). Thus, if the language of a domain theory has n

different actions terms, then $\mathcal{D}_{ap}$ consists of the following n axioms:

$$(\forall \vec{x}, s).Poss(A_1(\vec{x}), s) \equiv \Pi_{A_1}(\vec{x}, s),$$
$$\vdots$$
$$(\forall \vec{z}, s).Poss(A_n(\vec{z}), s) \equiv \Pi_{A_n}(\vec{x}, s),$$

where all formulas $\Pi_{A_i}(\vec{x}, s)$ are uniform in s.

Successor state axioms $\mathcal{D}_{ss}$. For each relational fluent $F(\vec{x}, s)$ there is one axiom

$$F(\vec{x}, do(a, s)) \;\equiv\; \Phi_F(\vec{x}, a, s),$$

where $\Phi_F(\vec{x}, a, s)$ is an uniform formula with free variables among $a, s, \vec{x}$. The successor state axioms characterize the truth values of the fluent F in the next situation $do(a, s)$ in terms of the logical conditions on current situation s. These axioms embody a solution to the frame problem for deterministic actions [Reiter, 1991]. In particular, the right hand side of each axiom mentions only actions that can cause the fluent F to become true and actions that can cause the fluent F to become false, all remaining actions have no effect on F:

$$F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s), \tag{2.6}$$

where $\gamma_F^+(\vec{x}, a, s)$ is a uniform situation calculus formula characterizing all conditions when an action a can make the fluent $F(\vec{x}, do(a, s))$ true in the situation $do(a, s)$, and $\gamma_F^-(\vec{x}, a, s)$ has the opposite meaning: it is a uniform situation calculus formula characterizing all conditions when an action a can make the fluent $F(\vec{x}, do(a, s))$ false in the situation $do(a, s)$. The solution to the frame problem suggested in [Reiter, 1991] is obtained under certain assumptions about the set of effect axioms which are provided by an axiomatizer. More specifically, it is supposed that for each fluent F, the axiomatizer provides only two types of axioms for F: **positive** normal form effect axiom

$$\gamma_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s))$$

and **negative** normal form effect axiom for fluent F:

$$\gamma_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s)).$$

Note that the axiomatization cannot include ramification constraints that constrain the indirect effects of actions and have the syntactic form

$$\begin{aligned} \upsilon_F^+(\vec{x}, a, s) &\supset F(\vec{x}, s) \\ \upsilon_F^-(\vec{x}, a, s) &\supset \neg F(\vec{x}, s) \end{aligned} \tag{2.7}$$

(e.g., either an application domain has no ramification constraints, or the axiomatizer has already managed 'to compile' all constraints into suitable effect axioms). In addition, it is supposed that the domain theory is consistent, in other words, the formula $\neg(\exists \vec{x}, a, s).\gamma_F^+(\vec{x}, a, s) \wedge$

$\gamma_F^-(\vec{x}, a, s)$ is entailed by the axiomatization (i.e., one cannot derive both $F(\vec{X}, do(A, S))$ and $\neg F(\vec{X}, do(A, S)))$. Finally, it is supposed that the set of these effect axioms complies with the following

Causal Completeness Assumption:

The positive and negative effect axioms characterize all the conditions under which action a causes fluent F to become true (respectively, false) in the successor situation.

According to [Reiter, 1991, Reiter, 2001a], one can prove from these assumptions, that the set of effect axioms is logically equivalent to the set of successor state axioms (2.6), one axiom for each fluent F. In the case when the first assumption is violated and the given axiomatization includes not only effect axioms, but also ramification constraints, there is a syntactic manipulation procedure that transforms the given set of effect axioms and ramification constraints (2.7) into a set of successor state axioms. This procedure and the proof of its correctness are formulated in [McIlraith, 2000] under the assumption that fluents can be separated into a partition and axioms can be stratified into ordered sequence excluding dependencies of fluents in the bottom layers from fluents that occur in the upper layers (this syntactic restriction on the set of effect and ramifications axioms often holds in interesting practical applications). In the example (2.1.1) below we will see the set of successor state axioms that includes certain ramification constraints in a 'compiled' form.

One can prove several interesting entailments from successor state axioms and foundational axioms Σ (these consequences are formulated in [Reiter, 2001a]). First,

$$\begin{aligned}
&F(\vec{x}, s) \equiv \\
&\qquad F(\vec{x}, S_0) \wedge \neg(\exists a, s')[do(a, s') \sqsubseteq s \wedge \gamma_F^-(\vec{x}, a, s')] \vee \\
&\qquad (\exists a', s')[do(a', s') \sqsubseteq s \wedge \gamma_F^+(\vec{x}, a', s') \wedge \\
&\qquad\qquad \neg(\exists a'', s'')[do(a', s') \sqsubset do(a'', s'') \sqsubseteq s \wedge \gamma_F^-(\vec{x}, a'', s'')]] \ ;
\end{aligned}$$

informally this means that the fluent F holds in situation s if and only if one of the following two conditions applies: either the fluent F holds in the initial situation S_0 and there was no action executed before s that caused F to become false, or there was a previous action a' that caused F to become true in a past situation s' and no subsequent action reversed this effect in any situation before s. Second, "closed form" solution for F:

$$\begin{aligned}
&F(\vec{x}, s) \equiv \ s = S_0 \wedge F(\vec{x}, S_0) \vee \\
&(\exists a, s')[\gamma_F^+(\vec{x}, a, s') \wedge s = do(a, s')] \vee \\
&(\exists s')[s' \sqsubseteq s \wedge F(\vec{x}, s') \wedge \neg(\exists a, s'')\{\gamma_F^-(\vec{x}, a, s'') \wedge s' \sqsubset do(a, s'') \sqsubseteq s\}] \ ;
\end{aligned}$$

intuitively, this means that the fluent F holds in situation s if and only if one of the following three conditions applies: s is S_0 and the fluent F holds initially, or the fluent F was caused to hold by the very last action, or there was a situation s' prior to s when F was true and no subsequent action executed before s caused F to become false.

As for functional fluents, their evolution from the current situation to the next situation is governed by similar axioms. For each functional fluent $f(\vec{x}, s)$ there is one axiom that has syntactic form $f(\vec{x}, do(a, s)) = y \;\equiv\; \phi_f(\vec{x}, y, a, s)$, where $\phi_f(\vec{x}, y, a, s)$ is a formula uniform in s that may have only $a, s, \vec{x}$ as free variables and such that the term f has a unique value:

$$\begin{aligned}\mathcal{D}_{una} \cup \mathcal{D}_{S_0} \models (\forall \vec{x}).(\exists y)\phi_f(\vec{x}, y, a, s) \land \\ [(\forall y, y').\phi_f(\vec{x}, y, a, s) = \phi_f(\vec{x}, y, a, s) \supset y = y'].\end{aligned}$$

The just mentioned *functional fluent consistency property* states that ϕ_f actually defines a value for f and that this condition is unique. Thus, if the language of a domain axiomatization includes m predicate fluents and k functional fluents, then $\mathcal{D}_{ss}$ is the set of the following axioms:

$$\begin{aligned} F_1(\vec{x}, do(a, s)) \;&\equiv\; \Phi_{F_1}(\vec{x}, a, s), \\ &\vdots \\ F_m(\vec{x}, do(a, s)) \;&\equiv\; \Phi_{F_m}(\vec{x}, a, s), \\ f_1(\vec{x}, do(a, s)) = y \;&\equiv\; \phi_{f_1}(\vec{x}, y, a, s), \\ &\vdots \\ f_k(\vec{x}, do(a, s)) = y \;&\equiv\; \phi_{f_k}(\vec{x}, y, a, s), \end{aligned}$$

Because right hand sides of all these formulas are uniform in s, values of fluents in the successor situation depend on logical conditions applied only in s, not in a predecessor of s or in a situation for which s is a predecessor. Because dynamical systems where the next state of a system depends only on the current state (and does not depend on past or future states) are called Markovian, the successor state axioms introduced in this section can be understood as defining a *Markovian dynamical system*.

A basic action theory is the following set of axioms:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ss} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}, \quad \text{where}$$

- Σ are the foundational axioms for situations.
- $\mathcal{D}_{ss}$ is a set of successor state axioms.
- $\mathcal{D}_{ap}$ is a set of action precondition axioms.

- $\mathcal{D}_{una}$ is the set of unique names axioms for actions.
- $\mathcal{D}_{S_0}$ is a set of first order sentences about the initial situation S_0.

Example 2.1.1: A Blocks World

The blocks world consists of a finite set of blocks located either somewhere on a table or on top of each other: any number of blocks can be on the table, but only one block can be on top of another block. In addition, there is a manipulator that can move a block from one location to another. In particular, it can move a block from the table on top of another block, or it can move a block from its current location to the table provided this block is not on the table already and there is nothing on the top of this block. In this example, we also make two common assumptions. First, all moving actions are deterministic, i.e., whenever the manipulator moves a block, it always succeeds: in the result of the action, the block will be at its destination, but not somewhere else. Second, there are no other agents (people, robots, manipulators, etc), who can move blocks.

The blocks world is axiomatized with successor state and action precondition axioms. We use the following function and predicate constants.

Actions

- $move(x, y)$: Move block x onto block y, provided both are clear.
- $moveToTable(x)$: Move block x onto the table, provided x is clear and is not on the table.

Fluents

- $On(x, y, s)$: Block x is on block y, in situation s.
- $Clear(x, s)$: Block x has no other blocks on top of it, in situation s.
- $Ontable(x, s)$: Block x is on the table in s.

Successor state axioms

$$\begin{aligned} On(x, y, do(a, s)) \equiv\ & a = move(x, y) \lor \\ & On(x, y, s) \land\ a \neq moveToTable(x) \land \neg(\exists z) a = move(x, z). \end{aligned}$$

$$Ontable(x, do(a, s)) \equiv\ a = moveToTable(x) \lor\ Ontable(x, s) \land \neg(\exists y) a = move(x, y).$$

$$\begin{aligned} Clear(x, do(a, s)) \equiv\ & (\exists y, z).[\, a = move(y, z) \lor\ a = moveToTable(y))\,] \land On(y, x, s) \lor \\ & Clear(x, s) \land \neg(\exists w) a = move(w, x). \end{aligned}$$

Action precondition axioms

$$Poss(move(x,y),s) \equiv\ Clear(x,s) \wedge\ Clear(y,s) \wedge x \neq y.$$

$$Poss(moveToTable(x),s) \equiv\ Clear(x,s) \wedge\ \neg Ontable(x,s).$$

Unique names axioms for actions

$$move(x,y) \neq moveToTable(x)$$

$$move(x,y) = move(x',y') \supset x = x' \wedge y = y'$$

$$moveToTable(x) = moveToTable(x') \supset x = x'$$

Assume that the following five ramification constraints $C(s)$ hold in initial situation S_0:

$$\begin{array}{l}
(\forall x,y).\ On(x,y,s) \supset \neg On(y,x,s) \\
(\forall x,y,z).\ On(y,x,s) \wedge On(z,x,s) \supset y = z \\
(\forall x,y,z).\ On(x,y,s) \wedge On(x,z,s) \supset y = z \\
(\forall x).\ Ontable(x) \equiv \neg(\exists y)\, On(x,y,s) \\
(\forall x).\ Clear(x,s) \equiv \neg(\exists y)\, On(y,x,s)
\end{array}$$

As mentioned in [Reiter, 2001a], one can prove that in this case the domain axiomatization $\mathcal{D}$ entails

$$(\forall s).reachable(s) \supset C(s),$$

where $reachable(s)$ is defined in (2.5). This means, that if $\mathcal{D}_{S_0}$ includes (or entails) five given constraints $C(S_0)$, then the axiomatization of the blocks world does not need to include aforementioned ramification constraints if we are interested only in entailments about reachable situations (in other words, indirect effects of actions are already predicted by the axiomatization).

2.1.2 The Sequential, Temporal Situation Calculus

In this subsection we consider adding an explicit representation for time into the basic situation calculus of previous Section 2.1.1. This representation will allow us to map a sequence of actions (leading to a certain situation) to a sequence of the exact times, or a range of times, at which actions are executed.

Actions

Actions are first-order terms consisting of an action function symbol and its arguments. In the sequential temporal situation calculus, the last argument of an action function symbol is the time of the action's occurrence that denotes the actual time at which that action occurs. Hence, according to this representation, all actions are instantaneous. For example, $startGo(l, l', 7.2)$ might denote the action of a robot starting to move from location l to l' at time 7.2 and term $endGo(l, l', 49.7)$ might denote the action of a robot ending to move at location l' at time 49.7. A new function symbol is introduced in the language: $time(a)$, denoting the occurrence time of action a. For each action term a, the set of axioms characterizing $time(a)$ as temporal argument of a is added to any basic action theory; for example:

$$(\forall l, l', t).time(startGo(l, l', t)) = t$$

Temporal variables range over the reals, although nothing prevents them from ranging over the rationals, or anything else on which a dense linear order relation $<$ can be defined. For this reason, axioms for the reals are not included in the axiomatization; it is assumed that temporal variables have the standard interpretation of the reals and we can rely on standard interpretations of operations (addition, multiplication, etc) and relations ($<$, $\leq$, etc).

Besides instantaneous actions, processes that have durations can be captured, as shown below.

Situations

A new function symbol is introduced in the language: $start(s)$, denoting the start time of situation s. This new symbol is characterized by:

$$start(do(a, s)) = time(a), \tag{2.8}$$

i.e. the start time of a situation is the occurrence time of the last action that led to this situation. The start time of S_0 can be defined arbitrarily, and may (or may not) be specified to be any number, depending on the application.

Foundational axioms (2.1)–(2.4) for situations without time are given in the previous subsection. Foundational axioms for situations with time are (2.1)–(2.4) and (2.8).

Because $s_1 \sqsubseteq s_2$ means that one can get to s' from s by a sequence of actions, the definition of reachability (2.5) has to be changed to make sure that s is reachable if and only if all actions leading to s are possible, and moreover, the times of action occurrences are not decreasing:

$$reachable(s) \stackrel{def}{=} (\forall a, s').\ do(a, s') \sqsubseteq s \supset Poss(a, s') \wedge start(s') \leq time(a). \tag{2.9}$$

Fluents

In the language of the temporal situation calculus, fluents represent properties of an application domain similarly to how we understand them in the basic situation calculus. In addition to this previously introduced understanding, we can understand fluents as representing properties of processes extended over time; those processes are delimited by two instantaneous actions: one of them initiate a process, another one terminates the process. For example, one may represent the process of going between two locations loc and loc' by the relational fluent $going(loc, loc', s)$. This process can be started by executing the initiating action *startGo*(l, l', t_1) in a situation preceding s and (if the process was started) it can be finished by executing the terminating action $endGo(l, l', t_2)$ in a situation for which s is a predecessor. As another example, one may represent a process fluent $moving(x, y, t, t', s)$, meaning that block x is in the process of moving to y, and t is initiation time of this process, t' is termination time. In this second case, one may introduce in the language instantaneous initiating and terminating actions, $startMove(x, y, t)$ and $endMove(x, y, t)$, with the obvious meanings.

Thus, if we compare fluents in the language of the temporal sequential situation calculus with fluents considered in the language of the basic situation calculus, then we may consider two alternatives: either there are no differences between representations of fluents (but there are some differences in how we understand fluents informally), or fluents can include additional temporal arguments corresponding to initiation (termination) times of a processes represented by them.

Basic Action Theories

Similarly to the basic situation calculus, a domain theory is axiomatized in the temporal situation calculus with the same set of axioms

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ss} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$$

supplemented with axioms about the functional symbol $time(a)$. We provide two examples.

Example 2.1.2: A Coffee Delivery Robot. We include from [Reiter, 1998, Reiter, 2001a] an axiomatization of the coffee delivery example in the temporal situation calculus (this example will be used also in subsequent sections and chapters). Imagine a robot whose task is to deliver coffee in an office environment. The robot is given a schedule of the preferred coffee periods of every employee, as well as information about the times it takes to travel between various locations in the office. The robot can carry just one cup of coffee at a time, and there is a central coffee machine from which it gets the coffee. Its task is to schedule coffee deliveries such that, if possible, everyone gets coffee during his/her preferred time periods. To simplify the axiomatization, we assume that the actions of *picking up* a cup of coffee and *giving it* to someone are instantaneous. The action of going from one location to another, which intuitively does have a duration, is represented by a pair of instantaneous $startGo$ and $endGo$ actions. In Golog procedures, CM (constant that denotes the coffee machine's location), and Sue, $Mary$, $Bill$, Joe (constants that denote people) always appear in upper case.

Primitive actions:

- $pickupCoffee(t)$. The robot picks up a cup of coffee from the coffee machine at time t.
- $giveCoffee(person, t)$. The robot gives a cup of coffee to $person$ at time t.
- $startGo(loc_1, loc_2, t)$. The robot starts to go from location loc_1 to loc_2 at time t.
- $endGo(loc_1, loc_2, t)$. The robot ends its process of going from loc_1 to loc_2 at time t.

Fluents:

- $robotLocation(s)$. A functional fluent denoting the robot's location in situation s.
- $going(loc_1, loc_2, s)$. This relational fluent means that in situation s, the robot is going from loc_1 to loc_2.
- $hasCoffee(person, s)$. $person$ has coffee in s.
- $holdingCoffee(s)$. This relational fluent means that in situation s, the robot is holding a cup of coffee.

Situation Independent Predicates and Functions

- $wantsCoffee(person, t_1, t_2)$. $person$ wants to receive coffee at some point in the time period $[t_1, t_2]$.
- $office(person)$. Denotes the office of $person$.
- $travelTime(loc_1, loc_2)$. Denotes the amount of time that the robot takes to travel between loc_1 and loc_2.
- Sue, $Mary$, $Bill$, Joe are constants denoting people; CM is the constant denoting coffee machine's location.

Action precondition axioms:

$$Poss(startGo(loc_1, loc_2, t), s) \equiv \neg(\exists l, l')going(l, l', s) \wedge \\ loc_1 \neq loc_2 \wedge robotLocation(s) = loc_1,$$

$$Poss(endGo(loc_1, loc_2, t), s) \equiv going(loc_1, loc_2, s).$$

$$Poss(pickupCoffee(t), s) \equiv \neg holdingCoffee(s) \wedge robotLocation(s) = CM,$$

$$Poss(giveCoffee(person, t), s) \equiv holdingCoffee(s) \wedge robotLocation(s) = office(person).$$

The last axiom is saying that the robot can give coffee to a person at the moment of time t in s iff the robot is holding a cup coffee (e.g., on a tray) and the robot is located in the office of the person.

Successor state axioms:

$$robotLocation(do(a, s)) = loc \equiv \\ (\exists t, loc')\, a = endGo(loc', loc, t) \vee \\ robotLocation(s) = loc \wedge \neg(\exists t, loc', loc'')\, a = endGo(loc', loc'', t),$$

Informally speaking, according to this axiom the robot will stay at a location until another action $endGo(loc', loc, t)$ will be executed, i.e., $startGo(loc', loc, t)$ actions are irrelevant and they do not change the location.

$$going(l, l', do(a, s)) \equiv (\exists t)a = startGo(l, l', t) \vee going(l, l', s) \wedge \neg(\exists t)a = endGo(l, l', t),$$

In English, the robot is going from one location to another if there exist a moment of time when the robot started to go between these two locations or the the robot was in the process of going in the previous situation and the most recent action did not terminated this process at time t.

$$hasCoffee(person, do(a, s)) \equiv (\exists t)a = giveCoffee(person, t) \vee hasCoffee(person, s),$$

$$holdingCoffee(do(a, s)) \equiv (\exists t)a = pickupCoffee(t) \vee \\ holdingCoffee(s) \wedge \neg(\exists p, t)\, a = giveCoffee(p, t).$$

Initial Situation

$$robotLocation(S_0) = CM,\ \neg(\exists l, l')going(l, l', S_0),\ start(S_0) = 0, \\ \neg holdingCoffee(S_0),\ \neg\exists p\, hasCoffee(p, S_0).$$

We have also a set of unique names axioms stating that the following terms are pairwise unequal: $Sue, Mary, Bill, Joe, CM$, $office(Sue)$, $office(Mary)$, $office(Bill)$, $office(Joe)$.

Coffee delivery preferences. The following expresses that $(Sue, 140, 160), \ldots, (Joe, 90, 100)$ are all, and only, the tuples in the $wantsCoffee$ relation.

$$wantsCoffee(p, t_1, t_2) \equiv \; p = Sue \wedge t_1 = 140 \wedge t_2 = 160 \vee$$
$$p = Mary \wedge t_1 = 130 \wedge t_2 = 170 \vee$$
$$p = Bill \wedge t_1 = 100 \wedge t_2 = 110 \vee p = Joe \wedge t_1 = 90 \wedge t_2 = 100.$$

The following set of axioms specifies how much time it takes for the robot to travel between two different locations[2]:

$$travelTime(CM, office(Sue)) = 15,$$
$$travelTime(CM, office(Mary)) = 10,$$
$$travelTime(CM, office(Bill)) = 8,$$
$$travelTime(CM, office(Joe)) = 10.$$
$$travelTime(l, l') = travelTime(l', l),$$
$$travelTime(l, l) = 0.$$

As we will see later in Example 2.3.2 and in Section 4.4.2, these values will constrain the time of $endGo$ actions.

Action Occurrence Times:

$$time(pickupCoffee(t)) = t, \; time(giveCoffee(p, t)) = t$$
$$time(startGo(l_1, l_2, t)) = t, \;\; time(endGo(l_1, l_2, t)) = t.$$

We will consider variations of this example in subsequent chapters.

2.1.3 The Situation Calculus with Knowledge

This variation of the situation calculus makes a distinction between *non-informative* (physical) actions that do not provide an agent with new information and *informative* actions that can contribute only to knowledge of an agent who executes them. Moore defines an action informative "if an agent would know more about the situation resulting from his performing the action after performing it than before performing it" (see [Moore, 1985], p. 350).

Fluents

The language of the situation calculus with knowledge may include a finite number of relational (functional) fluents, similarly to the language of the basic situation calculus. In addition, the

[2] For simplicity, we assume here that all these times are fixed numbers, but in the reality the travel times will vary. We address this issue in Chapter 4 when we consider replanning and rescheduling in Section 4.4.3.

former language has a special fluent that accounts for changes in an agent's beliefs caused by informative actions that the agent can execute. Following [Moore, 1985, Scherl and Levesque, 1993], the fact that situation s is one of the situations that the agent in situation σ considers possible is represented by $K(s, \sigma)$, where K is a binary predicate symbol that corresponds to an accessibility relation in modal logics of knowledge. Let ϕ be a situation-suppressed expression obtained from an uniform situation calculus formula $\phi(s)$ by suppressing all situation arguments in relational (functional) fluents mentioned in $\phi(s)$.[3] Then the fact that the agent knows ϕ in situation σ is defined by an abbreviation

$$\mathbf{Knows}(\phi, \sigma) \stackrel{def}{=} (\forall s).\, K(s, \sigma) \supset \phi[s]$$

(where $\phi[s]$ denotes that situation calculus formula obtained from ϕ by restoring the situation argument to its rightful place, see the definition (2.2.8) for details). The fact that an agent knows whether ϕ is true in situation σ is defined by an abbreviation

$$\mathbf{KWhether}(\phi, \sigma) \stackrel{def}{=} \mathbf{Knows}(\phi, \sigma) \vee \mathbf{Knows}(\neg\phi, \sigma);$$

the fact that an agent knows the referent of the term t is defined by an abbreviation

$$\mathbf{KRef}(t, \sigma) \stackrel{def}{=} (\exists x)\mathbf{Knows}(x = t, \sigma).$$

Note that the above definitions express an informal idea that the smaller is the set of situations s epistemically alternative to an actual situation σ, the more facts an agent knows; in particular, if σ is the only one situation alternative to σ, then the agent knows everything that is true in the actual world.

Actions

The language of the situation calculus with knowledge may include a finite number of action terms that are understood as physical actions. None of these actions may influence the $K(s, \sigma)$ fluent:

$$K(s', do(a, s)) \equiv (\exists s^*).s' = do(a, s^*) \wedge K(s^*, s). \tag{2.10}$$

Informally, this axiom means that physical actions do not provide the agent who executes them with new information: for any situation s^* accessible from an actual situation s, the situation $do(a, s^*)$ is accessible from $do(a, s)$, in other words, the number of alternative situations does

[3]See the definition (2.2.7) for a formal definition of the situation-suppressed formulas.

not change (but because all epistemically alternative situations result from executing an action a, the agent will know that the action a has been performed).

In addition to non-informative actions, the language may include two disjoint set of informative actions as described below.

For each of finitely many relational fluents there is a *sense* action whose purpose is to determine the truth value of its fluent. For example, a sense action $sense_{hasCoffee}(pers)$, determines whether the fluent $hasCoffee(pers, s)$ is true in s. Let $F(\vec{x}, s)$ be a relational fluent, then the sense action that determines a value of this fluent is denoted by $sense_F(\vec{x})$, and there are axioms (see below) such that after an agent performs a $sense_F$ action, the agent will know whether F, i.e. that

$$(\forall s, \vec{x})\mathbf{KWhether}(F(\vec{x}), do(sense_F(\vec{x}), s)).$$

In the sequel, the language includes a fixed, finite repertoire of actions $sense_{F_1}, \ldots, sense_{F_m}$.

Similarly, for each of finitely many functional fluents there is a *read* action whose purpose is to determine the referent of its fluent. For example, a read action $read_{robotLoc}(x)$ determines the value of the functional fluent $robotLoc(s)$ in s. Let $f(\vec{x}, s)$ be a functional fluent, then the read action that determines its referent is $read_f(\vec{x})$, and if this action is axiomatized appropriately (see below), then by performing a $read_f$ action, an agent will come to know the referent of f, i.e. that

$$(\forall s, \vec{x})\mathbf{KRef}(f(\vec{x}), do(read_f(\vec{x}), s)).$$

In the sequel, suppose that the agent has a fixed, finite repertoire of read actions $read_{f_1}, \ldots, read_{f_n}$.

It is convenient to introduce the *no side-effects assumption* for informative actions. Let $sense_F$ be any sense action, G be any relational fluent (different from $K(s, \sigma)$), $read_f$ be any read action, and g be any functional fluent; then no side-effects assumption means that successor state axioms (together with other axioms) entail the following formulas:

$$\begin{aligned}
&(\forall \vec{x}, \vec{y}, s)G(\vec{x}, s) \equiv G(\vec{x}, do(sense_F(\vec{y}), s)),\\
&(\forall \vec{x}, \vec{z}, s)G(\vec{x}, s) \equiv G(\vec{x}, do(read_f(\vec{z}), s)),\\
&(\forall \vec{x}, \vec{y}, s)g(\vec{x}, s) = g(\vec{x}, do(sense_F(\vec{y}), s)),\\
&(\forall \vec{x}, \vec{z}, s)g(\vec{x}, s) = g(\vec{x}, do(read_f(\vec{z}), s)).
\end{aligned} \tag{2.11}$$

Note that in the case when sense (read) action in an application domain is understood as having effect on a fluent, one can introduce two different actions that will account for the effect of that sense (read) action: one of them will have effects only on regular fluents and another one will provide only sensory information. Then, the no-side effect assumption will be enforced.

Situations

There are additional initial situations that serve as epistemic alternatives to the *actual* initial situation S_0 [Lakemeyer and Levesque, 1998]. They are needed to account for formulas ϕ which the robot does not know in S_0; if all these alternative initial situations are K-related to S_0 and to one another and in some of those situations ϕ is true, but in others it is false, then according to the definition, $\neg Knows(\phi, S_0)$. Each initial situation has no predecessor, for this reason it is convenient to define all of them by an abbreviation: $Init(s) \stackrel{def}{=} \neg(\exists a, s')s = do(a, s')$.

Because the basic situation calculus has the only initial situation S_0, but there may be several different initial situations in the situation calculus with knowledge, [Lakemeyer and Levesque, 1998, Levesque and Lakemeyer, 2000] propose to replace the old induction axiom (2.2) by

$$\begin{array}{c}(\forall P).(\forall s)[Init(s) \supset P(s)] \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \\ \supset (\forall s)P(s).\end{array} \tag{2.12}$$

Finally, there is an axiom saying that only initial situations can be K-related to an initial situation:

$$K(s, s') \supset [Init(s) \equiv Init(s')]. \tag{2.13}$$

The new foundational axioms Σ^K for the situation calculus with knowledge consist of (2.1), (2.3), (2.4), (2.12) and (2.13).

Basic Action Theories

A domain theory is axiomatized in the situation calculus with knowledge by means of the four classes of axioms mentioned in Section 2.1.1 supplemented with a new set of axioms $\mathcal{K}_{Init}$ about the knowledge fluent over initial situations.

The basic action theory with knowledge is
$\mathcal{D}^K = \mathcal{D}^K_{ss} \cup \mathcal{D}^K_{ap} \cup \mathcal{D}^K_{S_0} \cup \mathcal{K}_{Init} \cup \Sigma^K \cup \mathcal{D}^K_{una}$,
where Σ^K is introduced above.

Unique names axioms for actions $\mathcal{D}^K_{una}$. These axioms state that all physical, sense and read actions of the domain are pairwise unequal.

Initial theory of the world $\mathcal{D}^K_{S_0}$. In general case, it may include arbitrary formulas about knowledge such as **Knows**(ϕ, S_0), **KRef**$(f(\vec{x}), S_0)$, **KWhether**(ψ, σ), **Knows**(**Knows**$(\psi), S_0)$, as well as other uniform situation calculus formulas whose only situation term is S_0.

Successor state axioms $\mathcal{D}^K_{ss}$. It includes a set of successor state axioms for relational (functional) fluents with the syntactic form shown in Section 2.1.1, and includes an axiom (2.14):

$$\begin{array}{l} K(s', do(a,s)) \equiv \\ \quad (\exists s^*).s' = do(a, s^*) \wedge K(s^*, s) \wedge \\ \qquad (\forall \vec{x}_1)[a = sense_{F_1}(\vec{x}_1) \supset F_1(\vec{x}_1, s^*) \equiv F_1(\vec{x}_1, s)] \wedge \cdots \wedge \\ \qquad (\forall \vec{x}_m)[a = sense_{F_m}(\vec{x}_m) \supset F_m(\vec{x}_m, s^*) \equiv F_m(\vec{x}_m, s)] \wedge \\ \qquad (\forall \vec{y}_1)[a = read_{f_1}(\vec{y}_1) \supset f_1(\vec{y}_1, s^*) = f_1(\vec{y}_1, s)] \wedge \cdots \wedge \\ \qquad (\forall \vec{y}_n)[a = read_{f_n}(\vec{y}_n) \supset f_n(\vec{y}_n, s^*) = f_n(\vec{y}_n, s)]]. \end{array} \tag{2.14}$$

Indeed, let s^* be a situation K-accessible from an actual situation s (i.e., $K(s^*, s)$) and suppose that sense action $sense_{F_i}(\vec{x}_i)$ determines that a relational fluent F_i is true. Then, to guarantee that F_i is true in all situations alternative to $do(sense_{F_i}(\vec{x}_i), s)$, it is necessary to remove situations $do(sense_{F_i}(\vec{x}_i), s^*)$ where $F_i(\vec{x}_i)$ is not true from being accessible to $do(sense_{F_i}(\vec{x}_i), s)$. Consequently, those and only those situations s' must be K-accessible from $do(sense_{F_i}(\vec{x}_i), s)$ which are successors of one of s^* situations alternative to s ($s' = do(sense_{F_i}(\vec{x}_i), s^*) \wedge K(s^*, s)$) and where $F_i(do(sense_{F_i}(\vec{x}_i), s^*))$ is true. This yields the following condition on relation K:

$$\begin{array}{l} F_i(do(sense_{F_i}(\vec{x}_i), s)) \supset [K(s', do(sense_{F_i}(\vec{x}_i), s)) \equiv \\ \qquad (\exists s^*).s' = do(sense_{F_i}(\vec{x}_i), s^*) \wedge K(s^*, s) \wedge F_i(do(sense_{F_i}(\vec{x}_i), s^*))]. \end{array}$$

From the no-side effects assumption (2.11), we have that

$$(\forall s)\, F_i(do(sense_{F_i}(\vec{x}_i), s)) \equiv F_i(\vec{x}_i, s),$$

so the above condition on K leads to:

$$\begin{array}{l} F_i(\vec{x}_i, s) \supset [K(s', do(sense_{F_i}(\vec{x}_i), s)) \equiv \\ \qquad (\exists s^*).s' = do(sense_{F_i}(\vec{x}_i), s^*) \wedge K(s^*, s) \wedge F_i(\vec{x}_i, s^*)]. \end{array}$$

Similarly, suppose that sense action $sense_{F_i}(\vec{x}_i)$ determines that a relational fluent F_i is false and repeat the previous argument:

$$\begin{array}{l} \neg F_i(\vec{x}_i, s) \supset [K(s', do(sense_{F_i}(\vec{x}_i), s)) \equiv \\ \qquad (\exists s^*).s' = do(sense_{F_i}(\vec{x}_i), s^*) \wedge K(s^*, s) \wedge \neg F_i(\vec{x}_i, s^*)]. \end{array}$$

Informally, this says that action $sense_{F_i}(\vec{x}_i)$ shrinks the relation $K(s^*, s)$ by leaving successors of only those s^* situations alternative to an actual situation s, which satisfy $\neg F_i(\vec{x}_i, s^*)$. The conjunction of the last two formulas is logically equivalent to

$$a = sense_{F_i}(\vec{x}_i) \supset \\ [K(s', do(a, s)) \equiv \\ (\exists s^*).s' = do(a, s^*) \wedge K(s^*, s) \wedge F_i(\vec{x}_i, s^*) \equiv F_i(\vec{x}_i, s)]. \tag{2.15}$$

This formula conforms with the intuition that action $sense_{F_i}(\vec{x}_i)$ provides information about F_i by decreasing the set of situations alternative to $do(sense_{F_i}(\vec{x}_i), s)$: situations $do(sense_{F_i}(\vec{x}_i), s^*)$ remain accessible from $do(sense_{F_i}(\vec{x}_i), s)$ if and only if they are successors of situations s^* which agree with s regarding the truth value of fluent F (i.e., $F_i(\vec{x}_i, s^*) \equiv F_i(\vec{x}_i, s)$).

By following similar line of reasoning, one can establish that for each read action

$$a = read_{f_j}(\vec{x}_j) \supset \\ [K(s', do(a, s)) \equiv \\ (\exists s^*).s' = do(a, s^*) \wedge K(s^*, s) \wedge f_j(\vec{x}_j, s^*) = f_j(\vec{x}_j, s)]. \tag{2.16}$$

Because the combination of axioms (2.10), (2.15) and (2.16) is logically equivalent to the successor state axiom (2.14) for K-fluent, it is the only axiom that one needs to reason about dynamics of knowledge in $\mathcal{D}^K$.

It is possible to obtain also the following more general successor state axiom, where each sense action $sense_{\psi_i}(\vec{x}_i), \quad i = 1, \ldots, m$, is associated with a condition $\psi_i(\vec{x}_i, s)$ whose truth value in situation s the action is designed to determine:

$$\begin{aligned} K(s, do(a, \sigma)) \equiv \ & (\exists s^*).s = do(a, s^*) \wedge K(s^*, \sigma) \ \wedge \\ & (\forall \vec{x}_1)[a = sense_{\psi_1}(\vec{x}_1) \supset \psi_1(\vec{x}_1, s^*) \equiv \psi_1(\vec{x}_1, \sigma)] \\ & \wedge \cdots \wedge \\ & (\forall \vec{x}_m)[a = sense_{\psi_m}(\vec{x}_m) \supset \psi_m(\vec{x}_m, s^*) \equiv \psi_m(\vec{x}_m, \sigma)] \end{aligned} \tag{2.17}$$

The right-hand sides of all other successor state axioms in $\mathcal{D}^K_{ss}$ can mention physical actions only and have the syntactic form characterized in Section 2.1.1. Hence, these axioms conform with the no side-effect assumption (2.11): if s is a current situation and a is a sense (read) action, then all fluents have the same values in $do(a, s)$ that they have in s.

Precondition axioms $\mathcal{D}^K_{ap}$. These axioms have the syntactic form

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s),$$

where $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$, whose only situation term is s and it can mention expressions of the form **Knows**(ϕ, s), **KRef**$(f(\vec{y}), s)$. This characterizes (via the formula $\Pi_A(\vec{x}, s)$) the conditions under which it is possible to execute physical or informative action $A(\vec{x})$ in situation s.

Initial accessibility relation axioms $\mathcal{K}_{Init}$.
The accessibility relation is reflexive in initial situations if

$$(\forall s).Init(s) \supset K(s, s),$$

and symmetric in initial situations if

$$(\forall s_1, s_2).Init(s_1) \wedge Init(s_2) \supset [K(s_1, s_2) \supset K(s_2, s_1)].$$

It is possible to prove that foundational axioms and the successor state axiom for (2.14) guarantee that provided the accessibility relation is reflexive (symmetric, respectively) in all initial situations, then it will have the same property in all situations [Scherl and Levesque, 2003, Reiter, 2001a]:

$$(\forall s).[\,Init(s) \supset K(s, s)\,] \;\supset\; (\forall \sigma).K(\sigma, \sigma). \qquad \text{(reflexivity)} \qquad (2.18)$$

$$(\forall s, s').\,[\,Init(s) \wedge Init(s') \supset [K(s, s') \supset K(s', s)]\,]$$
$$\supset\; (\forall \sigma, \sigma').K(\sigma, \sigma') \supset K(\sigma', \sigma). \qquad \text{(symmetry)}$$

Other possible restrictions on the accessibility relation (transitive, euclidean) in initial situations can also be "propagated" to all situations. In the sequel, we assume that $\mathcal{K}_{Init}$ includes the axiom about reflexivity of $K(s, s')$ in all initial situations.

2.2 Regression

Regression is a general (and often efficient) computational mechanism used to evaluate whether a situation calculus formula is entailed by a basic theory of actions [Waldinger, 1977, Pednault, 1986, Pirri and Reiter, 1999, Reiter, 2001a]. Before going into details on how to define the regression operator $\mathcal{R}$ formally, we recall the intuition expressed in the definition of the regression operator. Suppose that $F(\vec{x}, do(a, s)) \equiv \Psi(\vec{x}, a, s)$ is a successor state axiom characterizing the relational fluent atom F, where $\Psi(\vec{x}, a, s)$ is a situation calculus formula. If $\phi(do(A, S))$ mentions F, we can determine a logically equivalent formula ϕ' by substituting $\Psi(\vec{x}, A, S)$ for $F(\vec{x}, do(A, S))$ in ϕ. Because the formula ϕ' has no occurrences of $F(\vec{x}, do(A, S))$ involving the complex situation term $do(A, S)$, this formula is simpler than the original formula $\phi(do(A, S))$. Because the same operation can be repeated with other fluents in ϕ' until we have no occurrences of the situation term $do(a, s)$, as the result we obtain a simpler logical formula that mentions only S_0. Similarly, the regression operator eliminates $Poss$ atoms in favor of their definitions as given by action precondition axioms.

The regression operator is defined not for arbitrary situation calculus formulas, but only for a large interesting class of formulas called *regressable*: these formulas occur naturally in planning and database query evaluation. An important simple example of regressable formulas is a situation calculus formula $\phi(S)$ that has the ground situation term S as the only situational argument, that is a first order formula built from relational fluents and equality atoms $t_1 = t_2$ (where t_1 and t_2 are not situation terms) by applying boolean operators and quantifiers over object variables.

Definition 2.2.1: The Regressable Formulas.
A situation calculus formula W is a regressable iff the following conditions hold

1. If a situation term occurs in W, then it has the syntactic form $do([\alpha_1, \ldots, \alpha_n], S_0)$ for some $n \geq 0$, and for terms $\alpha_1, \ldots, \alpha_n$ of sort action.

2. If an atom of the form $Poss(\alpha, \sigma)$ occurs in W, then α is a ground action term $A(t_1, \ldots, t_n)$ for some n-ary action function symbol A.

3. There are no occurrences of the predicate symbol $\sqsubset$ in W, nor does it mention any equality atom $s = s'$ for terms s, s' of sort situation.

This definition is less general than the definition given in [Pirri and Reiter, 1999], but it will be sufficient for our purposes and will simplify the following definitions.

Now suppose that W is a regressable situation calculus formula. The regression operator $\mathcal{R}_\mathcal{D}$ is designed to simplify W by replacing fluents and atoms that have occurrences of complex situation term $do([\alpha_1, \ldots, \alpha_n], S_0)$ by formulas with a simpler situation terms. The regression operator $\mathcal{R}_\mathcal{D}$ is determined relative to a basic theory of actions $\mathcal{D}$ that serves as a background axiomatization.

Definition 2.2.2: The Regression Operator.
Let W be a regressable formula, $\mathcal{D}$ be a basic theory of actions that includes a set of successor state axioms $\mathcal{D}_{ss}$ and a set of action precondition axioms $\mathcal{D}_{ap}$. Let $t_1, \ldots, t_n, x_1, \ldots, x_n, \vec{\tau}$ be terms of sort object, A, α, α' be terms of sort action, σ, σ' be terms of sort situation. The *regression operator* $\mathcal{R}_\mathcal{D}[\mathcal{W}]$ when applied to a formula W is defined inductively over the structure of

W (with the base case when W is an atom) as follows:[4]

1. When W is a non-fluent atom, including equality atoms (between terms of sort object or action), $\quad \mathcal{R}[W] = W$.

2. When W is a fluent atom whose situation argument is S_0, $\quad \mathcal{R}[W] = W$.

3. When W is a relational fluent atom $F(t_1, \ldots, t_n, do(\alpha, \sigma))$ whose successor state axiom in $\mathcal{D}_{ss}$ is

$$(\forall a, s, x_1, \ldots, x_n).F(x_1, \ldots, x_n, do(a, s)) \equiv \Phi_F(x_1, \ldots, x_n, a, s) \qquad (2.19)$$

then[5]

$$\mathcal{R}[\, F(t_1, \ldots, t_n, do(\alpha, \sigma))\,] \;=\; \mathcal{R}[\, \Phi_F|^{x_1,\ldots,x_n,a,s}_{t_1,\ldots,t_n,\alpha,\sigma}\,].$$

In other words, to regress the relational fluent, one has to replace it by a suitable instance of the formula representing the right hand side of the fluent's successor state axiom and regress this formula. To avoid potential conflicts between free variables (if any) of F with quantifiers (if any) of Φ_F, we can assume, without loss of generality, that if the formula $\Phi_F(x_1, \ldots, x_n, a, s)$ has any quantifiers, then their quantified variables have been renamed to be distinct from the free variables (if any) of $F(t_1, \ldots, t_n, do(\alpha, \sigma))$. This renaming prevents any of quantifiers of $\Phi_F(x_1, \ldots, x_n, a, s)$ from capturing variables in the instance $F(t_1, \ldots, t_n, do(\alpha, \sigma))$.

4. When W has an occurrence of a term $g(\vec{\tau}, do(\alpha', \sigma'))$, where g is a functional fluent, then W mentions a term of the form $f(t_1, \ldots, t_n, do(\alpha, \sigma))$ with the property that S_0 is the only situation term (if any) mentioned by $t_1, \ldots, t_n, \alpha$. Indeed, if W is a regressable formula, then one can prove (by induction on the sum of the length of all terms of sort situation that occur in $g(\vec{\tau}, do(\alpha', \sigma'))$) that $g(\vec{\tau}, do(\alpha', \sigma'))$ mentions a simple functional fluent $f(t_1, \ldots, \tau_n, do([\alpha_1, \ldots, \alpha_n], S_0))$ such that either $t_1, \ldots, t_n, \alpha_1, \ldots, \alpha_n$ have no occurrences of situation terms or they mention S_0 only.
Let f's successor state axiom in $\mathcal{D}_{ss}$ be

$$(\forall a, s, x_1, \ldots, x_n, y).f(x_1, \ldots, x_n, do(a, s)) = y \equiv \phi_f(x_1, \ldots, x_n, y, a, s) \qquad (2.20)$$

[4] We omit the subscript $\mathcal{D}$ because it is clear from the context that we talk about the regression operator defined with respect to the theory $\mathcal{D}$.

[5] In general, when ϕ is a formula, and t and t' are terms, then $\phi|^{t'}_{t}$ denotes the result of replacing all occurrences of t' (if any) in ϕ by t.

then

$$\mathcal{R}[\,W\,] \;=\; \mathcal{R}[\,(\exists y).\phi_f(t_1,\ldots,t_n,y,\alpha,\sigma) \wedge W|^{f(t_1,\ldots,t_n,do(\alpha,\sigma))}_{y}\,],$$

where y is a variable that does not occur free in $W, t_1, \ldots, t_n, \alpha, \sigma$. This says simply that to regress W, we have to replace W by a logically equivalent formula (using the right hand side of f's successor state axiom) and next regress this formula. Because

$$f(t_1,\ldots,t_n,do(\alpha,\sigma))$$

has an occurrence of a complex situation term $do(\alpha,\sigma)$, replacement yields a formula $(\exists y).\phi_f(t_1,\ldots,t_n,y,\alpha,\sigma) \wedge W|^{f(t_1,\ldots,t_n,do(\alpha,\sigma))}_{y}$ that has a shorter situation term. Similarly to the case of relational fluents, we assume that beforehand all quantified variables (if any) of $\phi_f(x_1,\ldots,x_n,y,a,s)$ have been renamed to be distinct from the free variables (if any) of $f(t_1,\ldots,t_n,do(\alpha,\sigma))$. This renaming makes sure that quantifiers in $\phi_f(x_1,\ldots,x_n,y,a,s)$ will not capture any variables in the instance of $f(t_1,\ldots,t_n,do(\alpha,\sigma))$. To complete the definition of the regression operator for functional fluents, note that if a regressable formula W mentions a term $g(\vec{\tau},do(\alpha',\sigma'))$, where g is a functional fluent, then W can have occurrences of several simple functional fluents of the form

$$f(t_1,\ldots,t_n,do(\alpha,\sigma)).$$

For this reason, to define $\mathcal{R}$ in a unique way, we can assume a suitable lexicographic ordering on functional fluent terms, and regress with respect to the least such term mentioned by W.

5. When W is a regressable atom $Poss(A(t_1,\ldots,t_n),\sigma)$ whose action precondition axiom in $\mathcal{D}_{ap}$ is

$$(\forall s,x_1,\ldots,x_n).Poss(A(x_1,\ldots,x_n),s) \equiv \Pi_A(x_1,\ldots,x_n,s) \tag{2.21}$$

then

$$\mathcal{R}[\,Poss(A(t_1,\ldots,t_n),\sigma)\,] \;=\; \mathcal{R}[\,\Pi_A|^{x_1,\ldots,x_n,s}_{t_1,\ldots,t_n,\sigma}\,]$$

In other words, to regress the atom $Poss$ that has an action function symbol A as an argument, one has to replace $Poss(A(t_1,\ldots,t_n),\sigma)$ by a suitable instance of the formula representing the right hand side of the action's precondition axiom and regress this formula. Similar to the previous case, one can rename (if necessary) all quantified variables of $\Pi_A(x_1,\ldots,x_n,s)$ to be distinct from the free variables (if any) of $Poss(A(t_1,\ldots,t_n),\sigma)$: renaming guarantees that quantifiers of $\Pi_A(x_1,\ldots,x_n,s)$ do not bound accidentally variables in the instance $Poss(A(t_1,\ldots,t_n),\sigma)$.

6. Whenever W is a formula,
   $\mathcal{R}[\neg W] = \neg\mathcal{R}[W]$,
   $\mathcal{R}[(\forall v)W] = (\forall v)\mathcal{R}[W]$,
   $\mathcal{R}[(\exists v)W] = (\exists v)\mathcal{R}[W]$.

7. Whenever W_1 and W_2 are formulas,
   $\mathcal{R}[W_1 \wedge W_2] = \mathcal{R}[W_1] \wedge \mathcal{R}[W_2]$,
   $\mathcal{R}[W_1 \vee W_2] = \mathcal{R}[W_1] \vee \mathcal{R}[W_2]$,
   $\mathcal{R}[W_1 \supset W_2] = \mathcal{R}[W_1] \supset \mathcal{R}[W_2]$,
   $\mathcal{R}[W_1 \equiv W_2] = \mathcal{R}[W_1] \equiv \mathcal{R}[W_2]$.

Thus, $\mathcal{R}[W]$ is simply that formula obtained from W by substituting repeatedly either suitable instances of the right hand side of successor state axiom for each occurrence of a fluent atom or suitable instances of the right hand side of action precondition axiom for each occurrence of a $Poss$ atom.

The idea behind the regression operator $\mathcal{R}$ is to reduce the depth of nesting of the function symbol do in the fluents of W by substituting suitable instances of Φ_F from (2.19) (or suitable instances of ϕ_f from 2.20, respectively) for each occurrence of a fluent atom of the form $F(t_1, \ldots, t_n, do(\alpha, \sigma))$ (for each occurrence of a functional fluent, respectively). Since no fluent of Φ_F (ϕ_f, respectively) mentions the function symbol do, the effect of this substitution is to replace each such $F(t_1, \ldots, t_n, do(\alpha, \sigma))$ (each such $f(t_1, \ldots, t_n, do(\alpha, \sigma))$, respectively) by a formula whose fluents mention only the situation term σ, and this reduces the depth of nesting by one. Therefore, the final result of regressing W is a formula whose only situation term is S_0, and because $\mathcal{R}$ substitutes logically equivalent expressions for atoms, the final formula is logically equivalent to W. The regression operator corresponds closely to the notion of *goal regression* in artificial intelligence planning procedures.

Example 2.2.3: Example of Regression. Let's consider the regressable formula

$$G = P(B, do(A, S_0)) \wedge Poss(A, S_0) \supset (\exists x).(P(x, S_0) \wedge x \neq B \vee Q(do(A, S_0))).$$

Here P and Q are fluents; equality atom $=$ is not a fluent. Suppose the successor state axioms for P and Q are

$$P(x, do(a, s)) \equiv \Phi_P(x, a, s),$$
$$Q(do(a, s)) \equiv \Phi_Q(a, s).$$

and suppose that the action precondition axiom for A is this: $Poss(A, s) \equiv \Pi_A(s)$. Then

$$\mathcal{R}[G] = \Phi_P(B, A, S_0) \wedge \Pi_A(S_0) \supset (\exists x).(\, P(x, S_0) \wedge x \neq B \vee \Phi_Q(A, S_0)\,).$$

The following important theorem establishes that the regression operator $\mathcal{R}$ is well-defined and also that the operator yields the formula logically equivalent to what it started with.

Theorem 2.2.4: *[Pirri and Reiter, 1999] Suppose W is regressable situation calculus formula and $\mathcal{D}$ is a basic theory of actions. Then $\mathcal{R}_{\mathcal{D}}[W]$ is a formula uniform in S_0. Moreover,*

$$\mathcal{D} \models (\forall)\,(W \equiv \mathcal{R}_{\mathcal{D}}[W]),$$

where $(\forall)\phi$ denotes the universal closure of the formula ϕ with respect to its free variables, and $\mathcal{R}_{\mathcal{D}}$ is the regression operator with respect to $\mathcal{D}$.

This theorem has a very important consequence: the task of establishing whether a regressable formula W is entailed by a basic theory of action $\mathcal{D}$ can be reduced to the task of establishing whether $\mathcal{R}_{\mathcal{D}}[W]$ is entailed by a small subset of $\mathcal{D}$; in particular, we do not need any axioms besides axioms about the initial situation and unique name axioms. Intuitively, this is true because actions precondition and successor state axioms no longer required as soon as we computed the final formula $\mathcal{R}[W]$. Moreover, none of the foundational axioms of Σ (including the second order induction axiom) are required, and this means that the entailment task can be reduced to a first order theorem proving task in $\mathcal{D}_{una} \cup \mathcal{D}_{S_0}$. The next theorem states this important result.

Theorem 2.2.5: (Soundness and Completeness of Regression) *[Pirri and Reiter, 1999] Suppose W is a regressable situation calculus sentence and $\mathcal{D}$ is a basic theory of actions. Then,*

1. *$\mathcal{R}_{\mathcal{D}}[W]$ is a sentence uniform in S_0*
2. *$\mathcal{D} \models \mathcal{R}_{\mathcal{D}}[W]$ iff $\mathcal{D}_{una} \cup \mathcal{D}_{S_0} \models \mathcal{R}_{\mathcal{D}}[W]$*

An important example of application of regression is the task of proving that a given action sequence is executable, i.e., that it leads to a reachable situation (see the formula 2.5). First, one can prove that given foundational axioms Σ, for each $n \geq 0$

$$\Sigma \models (\forall a_1, \ldots, a_n).\, reachable(do([a_1, \ldots, a_n], S_0)) \equiv \\ \bigwedge_{i=1}^{n} Poss(a_i, do([a_1, \ldots, a_{i-1}], S_0)).$$

Second, from this and regression theorem, one can prove (see [Reiter, 2001a])

Corollary 2.2.6: *Suppose that $\alpha_1, \ldots, \alpha_n$ is a sequence of ground action terms. Then*

$$\mathcal{D} \models reachable(do([\alpha_1, \ldots, \alpha_n], S_0))$$

iff

$$\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \bigwedge_{i=1}^{n} \mathcal{R}_{\mathcal{D}}[\, Poss(\alpha_i,\, do([\alpha_1, \ldots, \alpha_{i-1}], S_0))\,].$$

This corollary provides a systematic, regression-based method for determining whether a given ground situation $do([\alpha_1, \ldots, \alpha_n], S_0)$ is reachable, in other words, whether this ground situation corresponds to a sequence of actions in which it is really possible to perform actions one after the other. It is remarkable that this task can be reduced to a theorem-proving task in the initial database $\mathcal{D}_{S_0}$, together with unique names axioms for actions.

The regression operator $\mathcal{R}$ transforms a formula W to a logically equivalent formula with S_0 as the only situational term. It is convenient to define also a 'one-step' version of the regression operator. Given a regressable formula $\phi[do(\alpha, s)]$, a 'one-step' regression operator $\mathcal{R}^1(\phi, \alpha)$ computes a formula logically equivalent to $\phi[do(\alpha, s)]$ that may have occurrences of a simpler situation term s only. However, before giving a formal definition of $\mathcal{R}^1(\phi, \alpha)$, we need to define a large interesting sub-class of regressable formulas: $\mathcal{R}^1$ will be applied in the sequel only to formulas from this sub-class. All formulas of this sub-class can be constructed from (objective) situation-suppressed expressions. These expressions have been mentioned already (without a formal definition) in Section (2.1.3), they will be used also for the purposes of specifying the syntax of the high-level programming language Golog in Section (2.3). The next sequence of definitions follows [Pirri and Reiter, 1999, Reiter, 2001b] and introduces all required notions.

Definition 2.2.7: Situation-Suppressed Terms and Expression.

The *situation-suppressed terms* are inductively defined by:

1. Any variable of sort action or abject is a situation-suppressed term.
2. If f is an $n{+}1$-ary functional fluent, and $t_1, \ldots, t_n$ are situation-suppressed terms of sorts appropriate for the first n arguments of f, then $f(t_1, \ldots, t_n)$ is a situation-suppressed term.
3. If g is an m-ary non-functional fluent symbol other than S_0 or do, and $t_1, \ldots, t_m$ are situation-suppressed terms of sorts appropriate for the arguments of g, then $g(t_1, \ldots, t_m)$ is a situation-suppressed term.

The *situation-suppressed expressions* are inductively defined by:

1. Whenever t and t' are situation-suppressed terms of the same sort, then $t = t'$ is a situation-suppressed expression.[6]

2. When t is a situation-suppressed term of sort action, then $Poss(t)$ is a situation-suppressed expression.

3. When F is an $(n+1)$-ary relational fluent symbol and $t_1, \ldots, t_n$ are situation-suppressed terms of sorts appropriate for the first n arguments of F, then $F(t_1, \ldots, t_n)$ is a situation-suppressed expression.

4. When P is an m-ary non-fluent predicate symbol other than $\sqsubset$, and $t_1, \ldots, t_m$ are situation-suppressed terms of sorts appropriate for the arguments of P, then $P(t_1, \ldots, t_m)$ is a situation-suppressed expression.

5. When ϕ and ψ are situation-suppressed expressions, so are $\neg\phi$ and $\phi \wedge \psi$. When v is a variable of sort action or object, then $(\exists v)\phi$ is a situation-suppressed expression.

In brief, situation-suppressed expressions are first order, never quantify over situations, never mention $\sqsubset$, nor do they ever mention terms of sort situation.

Definition 2.2.8: Restoring Suppressed Situation Arguments

Whenever t is a situation-suppressed term and σ is a term of sort situation, $t[\sigma]$ denotes that term of the language obtained by restoring the term σ as the situation argument to all of the functional fluent terms mentioned by t. In addition, whenever ϕ is a situation-suppressed expression, $\phi[\sigma]$ denotes that formula of the language obtained by restoring the term σ as the situation argument to all of the functional fluent terms and all of the relational fluent atoms in ϕ.

Note that if $\phi(s)$ is a regressable situation calculus formula, ϕ is a situation-suppressed expression and S is a ground situation term, then $\phi[S]$ is a regressable formula.

Next, we can define a 'one-step' version of the regression operator that can be applied to $\phi[S]$, where ϕ is a situation-suppressed expression. To emphasize that a 'one-step' regression operator is different from the already introduced operator $\mathcal{R}$, we will denote this new operator

[6]Because situation-suppressed terms are never of sort situation, the situation-suppressed expressions never mention an equality atom between situations.

by a low-case letter ρ. (We omit the subscript $\mathcal{D}$, but all regression operators are defined with respect to the background theory $\mathcal{D}$.)

Definition 2.2.9: The Single Step Regression Operator

Let ϕ be a situation-suppressed expression, α be an action term, and $\phi[do(\alpha, s)]$ be a regressable situation calculus formula that has no occurrences of predicate symbol $Poss$. Then, $\rho^1(\phi, \alpha)$ is that situation-suppressed expression obtained by

- replacing all fluents in $\phi[do(\alpha, s)]$ with situation argument $do(\alpha, s)$ by the corresponding right-hand sides of their successor-state axioms (renaming quantified variables if necessary),
- and next, suppressing the situation arguments in the resulting formula.

By induction on the structure of the formula ϕ, one can prove that

Lemma 2.2.10:

$\mathcal{D} \models \rho^1(\phi, \alpha)[s] \equiv \phi[do(\alpha, s)]$,

in other words, $\rho^1(\phi, \alpha)[s]$ *is logically equivalent to* $\phi[do(\alpha, s)]$ *relative to the successor state axioms.*

By analogy with the single step regression operator $\rho^1(\phi, \alpha)$, it is convenient to introduce the 'multi step' regression operator defined on situation suppressed expressions.

Definition 2.2.11: The Multi Step Regression Operator

Let ϕ be a situation-suppressed expression, α be an action term, and $\phi[do(\alpha, s)]$ be a regressable situation calculus formula that has no occurrences of predicate symbol $Poss$. Then, $\rho(\phi, \sigma)$ is the situation-suppressed expression inductively defined by

1. $\rho(\phi, S_0) = \phi$
2. $\rho(\phi, do(\alpha, \sigma)) = \rho(\, \rho^1(\phi, \alpha), \sigma\,)$.

Note the difference between usages of ρ and $\mathcal{R}$ notations: the 1–argument regression operator $\mathcal{R}[W]$ takes a regressable formula W and denotes a logically equivalent situation calculus formula about S_0; the 2–argument operator $\rho(\phi, \sigma)$ represents regression of a situation-suppressed expression ϕ through the actions of σ.

The following theorem establishes the interesting connection between two notations:

Lemma 2.2.12: $\rho(\phi, \sigma)[S_0]$ *and* $\mathcal{R}[\, \phi[\sigma]\,]$ *are logically equivalent, relative to the basic action theory* $\mathcal{D}$*:*

$$\mathcal{D} \models \rho(\phi, \sigma)[S_0] \equiv \mathcal{R}[\, \phi[\sigma]\,].$$

2.2.1 The Projection Task

Regression is a key computational mechanism for solving the projection task, a common AI task that is a prerequisite to answering various other questions.

Informally, the forward projection task is this: given a sequence of ground action terms $\alpha_1, \ldots, \alpha_n$, and a formula $\phi(s)$ whose only free variable is the situation variable s, determine whether ϕ is true in the situation resulting from performing this action sequence, beginning with the initial situation S_0. This task can be formulated more precisely. Suppose that a sequence of ground action terms $\alpha_1, \ldots, \alpha_n$ mentions physical actions only. Let $\mathcal{T}$ be a basic action theory of the domain and let $\phi(do([\alpha_1, \ldots, \alpha_n], S_0))$ be a regressable situation calculus formula (where $\phi(s)$ is a situation calculus formula uniform in s).

The forward projection task is to determine whether

$$\mathcal{T} \models \phi(do([\alpha_1, \ldots, \alpha_n], S_0)).$$

In the case when $\mathcal{T}$ is a domain theory without knowledge and sensing, from Theorem (2.2.5) follows that regression is a sound (and under certain conditions computationally efficient) way of solving the forward projection task for regressable formulas.

More precisely, suppose that $\alpha_1, \ldots, \alpha_n$ is a sequence of ground action terms and that formula $\phi(do([\alpha_1, \ldots, \alpha_n], S_0))$ is a regressable situation calculus formula. In particular, let $\phi(s)$ be a situation calculus formula uniform in s and have no other free variables except for s. Then one can use regression to solve the forward projection task:

$$\mathcal{D} \models \phi(do([\alpha_1, \ldots, \alpha_n], S_0))$$

iff

$$\mathcal{D}_{una} \cup \mathcal{D}_{S_0} \models \mathcal{R}_{\mathcal{D}}[\, \phi(do([\alpha_1, \ldots, \alpha_n], S_0))\,],$$

where the subscript $\mathcal{D}$ indicates that the regression operator is defined relative to $\mathcal{D}$. Because in the projection task we are interested only in a special sub-class of regressable formulas, the regression operator defined for a larger class also can be applied to determine entailments.

2.3 GOLOG

Golog [Levesque *et al.*, 1997] is a situation calculus-based high level programming language for defining complex actions in terms of a set of primitive actions. Planning is known to be computationally intractable in general and is impractical for deriving complex behaviors involving hundreds, and possibly thousands of actions in applications characterized by hundreds of different fluents. For this reason, the Cognitive Robotics Group at the University of

Toronto examines a computer science approach: reduce the reliance on *planning* for eliciting interesting robot behaviors, and instead provide the robot with *programs* written in a suitable high-level language, in our case, Golog. As presented in [Levesque *et al.*, 1997] and extended in [De Giacomo *et al.*, 1997b], Golog is a logic-based programming language whose primitive actions are those of a background domain theory $\mathcal{D}$. All primitive actions are axiomatized in the situation calculus as described above.

Golog has the standard control structures found in most Algol-like languages — and some not so standard.

1. *Sequence:* $\alpha_1 \ ; \ \alpha_2$. Do action α_1, followed by action α_2.
2. *Test actions:* ϕ? Test the truth value of expression ϕ in the current situation.[7]
3. *Nondeterministic action choice:* $\alpha_1 \mid \alpha_2$. Do α_1 or α_2.
4. *Nondeterministic choice of arguments:* $(\pi\ x)\alpha$. Nondeterministically pick a value for x, and for that value of x, do action α.
5. *Conditionals:* **if** ϕ **then** α_1 **else** α_2. Evaluate the expression ϕ and if it is true then do α_1 else do α_2.
6. *while loops*: **while** ϕ **do** α. Evaluate the expression ϕ and if it is true then do α, otherwise do nothing.
7. *Procedures, including recursion.*

In the subsequent sections we discuss two different semantics for Golog: evaluation semantics and transition semantics.

2.3.1 Evaluation semantics

The evaluation semantics of Golog programs is defined by macro-expansion, using a ternary relation *Do*. $Do(\delta, s, s')$ is an *abbreviation* for a situation calculus formula whose intuitive meaning is that s' is a situation reached from situation s by one of the sequences of actions specified by the program δ (because δ may include nondeterministic operators the sequence of actions specified by δ is not necessarily unique). To determine this sequence of actions from δ, one *proves*, using the situation calculus axiomatization of the background domain $\mathcal{D}$, the formula $(\exists s) Do(\delta, S_0, s)$. Thus, it is assumed that a Golog programmer has certain intuitions

[7] It is important to understand that tests in Golog are distinguished from sensing: sensing measures values of fluents in the external world, but tests are evaluated with respect to domain axioms.

about the domain and is able to write a program δ such that any binding for the existentially quantified variable s obtained as a side effect of establishing the entailment

$$\mathcal{D} \models (\exists s)\, Do(\delta, S_0, s)$$

constitutes a suitable plan, in terms of the primitive actions, specified by δ. In his seminal work [Green, 1969, Green, 1980], Cordell Green proposed a deductive approach to planning using the situation calculus. In contrast to C.Green's approach to "planning by theorem proving", a programmer may use any of Golog constructs to write a program that constrains the search for a desirable plan (if a Golog program is deterministic, then no search is required to find a binding for the existentially quantified variable s). Note also that in contrast to straight line or partially ordered plans, a Golog program can be arbitrary complex, including loops, conditionals, recursive procedures and nondeterministic choices between branches.

Do is defined inductively on the structure of its first argument as follows:

1. **Primitive actions:**

$$Do(a, s, s') \stackrel{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s). \qquad (2.22)$$

   The notation $a[s]$ mean the result of restoring the situation argument s to all functional fluents mentioned by the action term a.

2. **Test actions:** $Do(\phi?, s, s') \stackrel{def}{=} \phi[s] \wedge s = s'$,
   where ϕ is a situation suppressed expression (see Definition 2.2.7) and $\phi[s]$ is the situation calculus formula obtained by restoring situation variable s to all fluent names in ϕ (see Definition 2.2.8).

3. **Sequence:** $Do(\delta_1; \delta_2, s, s') \stackrel{def}{=} (\exists s^*).Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s')$.

4. **Nondeterministic choice of two actions:** $Do(\delta_1 | \delta_2, s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$.

5. **Nondeterministic choice of action arguments:**

$$Do((\pi x)\, \delta(x), s, s') \stackrel{def}{=} (\exists x)\, Do(\delta(x), s, s').$$

6. **Nondeterministic iteration:**

$$\begin{aligned} &Do(\delta^*, s, s') \stackrel{def}{=} \\ &\quad (\forall P).\{(\forall s_1) P(s_1, s_1) \wedge (\forall s_1, s_2, s_3)[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\} \\ &\qquad \supset P(s, s'). \end{aligned}$$

7. **Conditionals:** **if** ϕ **then** δ_1 **else** δ_2 **endIf** $\stackrel{def}{=} [\phi?; \delta_1] | [\neg\phi?; \delta_2]$

8. **while-loops:** **while** ϕ **do** δ **endWhile** $\stackrel{def}{=} [\phi?; \delta]^* \neg\phi?$

9. **Procedures:** First, we need an auxiliary macro definition of a complex action that consists of procedure call P only. For any predicate variable P of arity $n+2$ that has situation variables as the last two arguments

$$Do(P(t_1, \ldots, t_n), s, s') \stackrel{def}{=} P(t_1[s], \ldots, t_n[s], s, s').$$

Expressions of the form $P(t_1, \ldots, t_n)$ serve in programs as procedure calls, and macro $Do(P(t_1, \ldots, t_n), s, s')$ means that executing the procedure P on actual parameters $t_1[s]$, ..., $t_n[s]$ causes a transition from situation s to s'.

Second, suppose that a program consists of procedures $P_1, \ldots, P_n$ with formal parameters $\vec{v_1}, \ldots, \vec{v_n}$ and procedure bodies $\delta_1, \ldots, \delta_n$ respectively, followed by a main program body δ_0. Here, $\delta_1, \ldots, \delta_n, \delta_0$ are complex actions expressions constructed from primitive actions, tests, procedure calls and other constructs defined above. The result of evaluating a program of this form

$$\textbf{proc } P_1(\vec{v_1})\delta_1 \textbf{ endProc} \,; \ldots \textbf{proc } P_n(\vec{v_n})\delta_n \textbf{ endProc} \,; \delta_0$$

is defined as follows:

$$\begin{aligned} &Do(\{\textbf{proc } P_1(\vec{v_1})\delta_1 \textbf{ endProc} \,; \ldots \textbf{proc } P_n(\vec{v_n})\delta_n \textbf{ endProc} \,; \delta_0\}, s, s') \stackrel{def}{=} \\ &\qquad (\forall P_1, \ldots, P_n).[\textstyle\bigwedge_{i=1}^{n} (\forall s_1, s_2, \vec{v_i}).Do(\delta_i, s_1, s_2) \supset P_i(\vec{v_i}, s_1, s_2)] \\ &\qquad\qquad \supset Do(\delta_0, s, s') \end{aligned}$$

Note that thanks to universal quantification over predicates $P_1, \ldots, P_n$ this macro expansion says that when $P_1, \ldots, P_n$ are the smallest binary relations on situations that are closed under the evaluation of their procedure bodies $\delta_1, \ldots, \delta_n$, then any sequence of actions from s to s' obtained by evaluating δ_0 is a sequence of actions for the evaluation of the given program.

In the sequel, we will consider Golog programs composed from primitive actions with an additional time argument. These actions are axiomatized in the temporal situation calculus that is overviewed in Section 2.1.2. To modify the Golog semantics in this case, we need to update only the first item (2.22) from the above inductive definition. This new definition of the Do macro for primitive actions is the following:

$$Do(a, s, s') \stackrel{def}{=} Poss(a, s) \wedge start(s) \leq time(a) \wedge s' = do(a, s).$$

Everything else about the definition of Do remains the same: $Do(program, s, s')$ is an *abbreviation* for a situation calculus formula whose intuitive reading is that s' is one of the situations reached from s by executing $program$. The modified definition of the Do macro makes sure that times of primitive actions leading from s to s' form non-decreasing sequence. To evaluate $program$, one must prove, using the situation calculus axiomatization of some background domain, the situation calculus formula $(\exists s)Do(program, S_0, s)$. Any binding for s obtained by a constructive proof of this sentence is an execution trace, in terms of the primitive actions, of $program$. Depending on the application domain, temporal terms of actions that occur in a binding for s can be constants or variables: if they are variables, this means that the application domain does not constrain completely times when actions have be executed. A Golog interpreter for the situation calculus with time, written in Prolog, is described in [Reiter, 1998].

Example 2.3.1: The following is a nondeterministic Golog program for the blocks world example. $makeOneTower(z)$ creates a single tower of blocks, using as a base the tower whose top block is initially z.

proc $makeOneTower(z)$
$\neg(\exists y).y \neq z \wedge clear(y)? \mid$
$(\pi\, x, t)[startMove(x, z, t)\ ;$
$\quad (\pi\, t')endMove(x, z, t')\ ;\ makeOneTower(x)]$
endProc

Like any Golog program, this is executed by proving

$$(\exists s)Do(makeOneTower(z), S_0, s),$$

in our case, using background axioms above. We start with S_0 as the current situation. In general, if σ is the current situation, $makeOneTower(z)$ terminates in situation σ if $\neg(\exists y).y \neq z \wedge clear(y, \sigma)$ holds. Otherwise it nondeterministically selects a block x and time t, and "performs" $startMove(x, z, t)$, meaning it makes $do(startMove(x, z, t), \sigma)$ the current situation; then it picks a time t' and "performs" $endMove(x, z, t')$, making

$$do(endMove(x, z, t'), do(startMove(x, z, t), \sigma))$$

the current situation; then it calls itself recursively. On termination, the current situation is returned as a side effect of the computation; this is an execution trace of the program. It is important to understand that this is an offline computation; the resulting trace is intended to be passed to some execution module for the online execution of the primitive actions in the program trace, in our example, to physically build the tower.

2.3.2 Transition semantics

This semantics is defined using axioms for *Trans* and *Final* that are proposed in [De Giacomo *et al.*, 1997b] and discussed in [De Giacomo *et al.*, 2000].[8] The predicate $\mathit{Trans}(\delta_1, s_1, \delta_2, s_2)$ holds if a Golog program δ_1 makes a transition in situation s_1 to a program δ_2 and results in situation s_2. Often, δ_2 is a structurally simpler program than δ_1, with an exception when δ_1 is a while-loop. The following set of axioms characterizes this predicate by induction on the structure of the program term δ_1:

1. Empty program: $\mathit{Trans}(\mathit{Nil}, s, \delta', s') \equiv \mathit{False}$

2. Primitive actions[9]:

$$\mathit{Trans}(a, s, \delta', s') \equiv \mathit{Poss}(a, s) \wedge \delta' = \mathit{Nil} \wedge s' = do(a, s) \wedge start(s) \leq time(a), \quad (2.23)$$

   where functional symbols $start(s), time(a)$ are defined in Section 2.1.2 and related by the equation (2.8).

3. Tests: $\mathit{Trans}(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = \mathit{Nil} \wedge s' = s$

4. Sequence:

$$\mathit{Trans}(\delta_1; \delta_2, s, \delta', s') \equiv \exists\gamma.\mathit{Trans}(\delta_1, s, \gamma, s') \wedge \delta' = \gamma; \delta_2 \ \vee \\ Final(\delta_1, s) \wedge \mathit{Trans}(\delta_2, s, \delta', s')$$

5. Nondeterministic choice:

$$\mathit{Trans}(\delta_1 | \delta_2, s, \delta', s') \equiv \mathit{Trans}(\delta_1, s, \delta', s') \vee \mathit{Trans}(\delta_2, s, \delta', s')$$

6. Conditional:

$$\mathit{Trans}(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta', s') \equiv \\ \phi[s] \wedge \mathit{Trans}(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge \mathit{Trans}(\delta_2, s, \delta', s')$$

7. Pick: [10] $\mathit{Trans}(\pi v.\delta(v), s, \delta', s') \equiv \exists x.\mathit{Trans}(\delta^v_x, s, \delta', s')$

[8] We omit axioms for procedures because they are too intricate, see [De Giacomo *et al.*, 2000] for details. We also omit axioms that define constructs included only in ConGolog.

[9] Because we are interested in temporal Golog programs, we consider a modified definition of *Trans* that is applicable in the case when all primitive actions have a temporal argument. The original semantics of *Trans* given in [De Giacomo *et al.*, 1997b] is defined for non-temporal situation calculus; the axiom for transition over a primitive action without a temporal argument is defined in that paper as follows:
$\mathit{Trans}(a, s, \delta', s') \equiv \mathit{Poss}(a, s) \wedge \delta' = \mathit{Nil} \wedge s' = do(a, s)$.

[10] Here, δ^v_x is the program resulting from substituting a fresh variable x for v uniformly in δ.

8. Iteration: $\mathit{Trans}(\delta^*, s, \delta', s') \equiv \exists\gamma.\mathit{Trans}(\delta, s, \gamma, s') \wedge \delta' = \gamma; \delta^*$

9. Loop:

$$\mathit{Trans}(\textbf{while } \phi \textbf{ do } \delta, s, \delta', s') \equiv \\ \exists\gamma.\mathit{Trans}(\delta, s, \gamma, s') \wedge \phi[s] \wedge (\delta' = \gamma; \textbf{while } \phi \textbf{ do } \delta)$$

Given a program δ and a situation s, $\mathit{Trans}(\delta, s, \delta', s')$ tells us which is a possible next step in the computation, returning the resulting situation s' and the program δ' that remains to be executed. In other words, $\mathit{Trans}(\delta, s, \delta', s')$ denotes a transition relation between configurations. Thus, the assertions above characterize when a configuration (δ, s) can evolve (in a single step) to a configuration (δ', s'). Note that conditional statements and loops can be defined from tests, nondeterministic choice and iteration.

The predicate $\mathit{Final}(\delta, s)$ holds if term δ is a program in a final state in situation s and either the execution of δ successfully terminated or the execution of δ is stuck (i.e., it cannot be successfully completed due to interference from exogenous actions). This predicate is inductively characterized by the following axioms:

1. Empty program: $\mathit{Final}(\mathit{Nil}, s) \equiv \mathit{True}$

2. Primitive action: $\mathit{Final}(a, s) \equiv \mathit{False}$

3. Test: $\mathit{Final}(\phi?, s) \equiv \mathit{False}$

4. Sequence: $\mathit{Final}(\delta_1; \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \wedge \mathit{Final}(\delta_2, s)$

5. Nondeterministic choice: $\mathit{Final}(\delta_1 | \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \ \vee \ \mathit{Final}(\delta_2, s)$

6. Conditional:

$$\mathit{Final}(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s) \equiv \phi[s] \wedge \mathit{Final}(\delta_1, s) \vee \neg\phi[s] \wedge \mathit{Final}(\delta_2, s)$$

7. Pick: $\mathit{Final}(\pi v.\delta, s) \equiv \exists x.\mathit{Final}(\delta^v_x, s)$

8. Iteration: $\mathit{Final}(\delta^*, s) \equiv \mathit{True}$

9. Loop: $\mathit{Final}(\textbf{while } \phi \textbf{ do } \delta, s) \equiv \mathit{Final}(\delta, s) \vee \neg\phi[s].$

This set of assertions characterizes when a configuration (δ, s) can be considered final, that is whether the computation is completed (no program remains to be executed).

There are also axioms for procedures, see [De Giacomo *et al.*, 2000] for details. We also omit axioms that define constructs included only in ConGolog.

In Chapter 4, to take into account programs that may get stuck we need also:

$$(\mathit{Trans}(Stop, s, \delta', s') \equiv \mathit{False}) \;\wedge\; (\mathit{Final}(Stop, s) \equiv \mathit{True}), \tag{2.24}$$

where $Stop$ is an auxiliary constant that denotes an abnormally terminated execution (the execution was aborted and the programmer must provide further instructions). Note that axioms for both *Nil* and $Stop$ are similar, but these two constant have different semantics: *Nil* represents the successfully completed execution, but $Stop$ represents abnormal termination. We introduce two different constants intentionally to distinguish between these two different cases when the execution terminates.

The possible configurations that can be reached by a program δ starting in a situation s are those obtained by following repeatedly the transition relation denoted by *Trans* starting from (δ, s), i.e. those in the reflexive transitive closure of the transition relation. Such a relation, denoted by *Trans**, is defined as the (second-order) situation calculus formula:

$$\mathit{Trans}^*(\delta, s, \delta', s') \equiv \forall T[\ldots \;\supset\; T(\delta, s, \delta', s')]$$

where ... stands for the conjunction of the universal closure of

$$T(\delta, s, \delta, s)$$
$$\mathit{Trans}(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \;\supset\; T(\delta, s, \delta', s')$$

Using *Trans** and *Final*, a definition of the *Do* relation of [Levesque *et al.*, 1997] is this:

$$\mathit{Do}(\delta, s, s') \equiv \exists \delta'.\mathit{Trans}^*(\delta, s, \delta', s') \wedge \mathit{Final}(\delta', s').$$

Recall that the relation $Do(\delta, s, s')$ means that s' is a terminating situation resulting from an execution of program δ beginning with situation s.

A straightforward implementation of this specification in Prolog is suggested in [De Giacomo *et al.*, 1997b, De Giacomo and Levesque, 1999b]. An incremental interpreter can work as follows. Given a terminating program δ and a situation s the interpreter determines whether the configuration (δ, s) is final. If yes, then a current substitution for s corresponds to a sought for plan; otherwise, if there is a transition *Trans*(δ, s, δ', s'), then (δ', s') is a next configuration. The computation continues incrementally until the interpreter arrives at a final configuration or until it fails if no final configuration can be achieved. From the axioms for *Trans* and *Final* it follows that given a Golog program γ, at every step of computation the interpreter selects either a next primitive action for execution or a next test condition for evaluation, until the program terminates or fails.

In the sequel, we use implemented Golog interpreters such that the domain specific action precondition and successor state axioms, and axioms about the initial situation, are expressible as Prolog clauses. Therefore, the implementation considered in this thesis inherits Prolog's closed world assumption, but this is a limitation of the implementation, not the general theory. We emphasize that it is not necessary to tie up an implementation of Golog with the closed world assumption. It is feasible to provide an implementation of Golog in the open world along the lines suggested in [Finzi *et al.*, 2000, Reiter, 2001a].

We reproduce below from [Reiter, 1998] a Golog program for a robot that can serve coffee requests to employees in an office-type environment (the version of this program in [Reiter, 2001a] has minor differences from the program discussed here). This Golog program relies on the example (2.1.2) that illustrates the basic theory of actions in Section 2.1.2 where all actions are instantaneous and have an explicit temporal argument.

Example 2.3.2: The top level Golog procedure $deliverCoffee(t)$ (with one temporal argument) is a program that terminates successfully when everyone who wants coffee has been given coffee. Otherwise, if there is at least one person who wants coffee and does not have it yet, the program calls the procedure $deliverOneCoffee(t)$ (when the robot is near the coffee machine), or sends the robot to the coffee machine and then the robot must execute the procedure $deliverOneCoffee(now)$ at the time of the arrival. The program $deliverOneCoffee(t)$ picks up nondeterministically an employee to be served, finds the travel time to the office of the employee from the coffee machine and chooses nondeterministically a wait interval, commands to pick up the coffee, go to the office of the employee and give coffee to a waiting person. Finally, the program calls recursively $deliverCoffee(now)$. This program uses two procedures.

proc $deliverCoffee(t)$

% Beginning at time t the robot serves coffee to everyone, if possible.

% Else (when the set of requests has no solution) the program fails.

$now \leq t?$;

{ $[(\forall p, t', t'').wantsCoffee(p, t', t'') \supset hasCoffee(p)]?$

$|$

if $robotLocation = CM$ **then** $deliverOneCoffee(t)$

else $goto(CM, t)$; $deliverOneCoffee(now)$

endIf }

endProc

proc $deliverOneCoffee(t)$

% Assuming the robot is located near the coffee machine,

% it delivers one cup of coffee.

$$
\begin{array}{l}
(\pi p, t_1, t_2, wait)[\ \{wantsCoffee(p, t_1, t_2)\ \wedge \neg hasCoffee(p) \wedge wait \geq 0\ \wedge \\
\qquad t_1 \leq t + wait + travelTime(CM, office(p)) \leq t_2\}?\ ; \\
\quad pickupCoffee(t + wait)\ ; \\
\quad goto(office(p), now)\ ; \\
\quad giveCoffee(p, now)\ ; \\
\quad deliverCoffee(now)]
\end{array}
$$

endProc

The above procedure introduces a functional fluent $now(s)$, which is identical to the fluent $start(s)$. We use it instead of $start$ because it has a certain mnemonic value, but, like $start$, it denotes the *current time*. The remaining two procedures are very simple. The first procedure commands the robot to go from the current position denoted by the functional fluent $robotLocation$ to location loc beginning at time t. The second commands the robot to go from the location $loc1$ to $loc2$ beginning at time t and taking Δ time units; it includes the sequence of $startGo$ and $endGo$.

proc $goto(loc, t)$ % Beginning at time t the robot goes to loc.

$$
\begin{array}{l}
goBetween(robotLocation, loc, \\
\qquad travelTime(robotLocation, loc), t)
\end{array}
$$

endProc

proc $goBetween(loc1, loc2, \Delta, t)$

% Beginning at time t the robot goes from loc1 to loc2,

% taking Δ time units for the transition.

$$
\begin{array}{l}
(loc1 = loc2 \wedge \Delta = 0)?\ | \\
[(loc1 \neq loc2 \wedge \Delta > 0)?\ ;\ startGo(loc1, loc2, t)\ ; \\
\qquad endGo(loc1, loc2, t + \Delta)]
\end{array}
$$

endProc

It was mentioned in Section 2.1.2, that each pair of adjacent actions in any reachable situation must satisfy the inequality (2.9) between start time of the current situation (determined by a previous action) and the occurrence time of the action to be executed (recall that we take $start(S_0) = 0$). When a *Trans*-based interpreter determines a transition for a primitive action it makes sure that a similar inequality determines a constraint on occurrence times of each pair of adjacent primitive actions. In a general case, this set of temporal constraints may have infinitely many solutions and the execution of the Golog call

$$\mathcal{D} \models \exists s. Do(deliverCoffee(1), S_0, s)$$

will not result in a fully instantiated sequence of actions S. The actions in that sequence will not have their occurrence times uniquely determined. Rather, these occurrence times will consist of all feasible solutions to the system of constraints generated by the program execution. So, to get a fixed schedule of coffee delivery, we can determine one or more of these feasible solutions, e.g., by minimizing the occurrence time with respect to the set of constraints.

2.3.3 On vs. Off-Line Golog Interpreters

Later in this thesis (in Chapter 4), when we describe our approach to execution monitoring, we distinguish carefully between on-line and off-line Golog interpreters. An on-line interpreter based on *Trans* and *Final* was originally proposed in [De Giacomo and Levesque, 1999b] to give an account of Golog/ConGolog programs with sensing actions. It was further employed in our paper [De Giacomo *et al.*, 1998] to define various versions of execution monitoring. Here we make use of a simplified on-line interpreter that does not deal with sensing actions. As we mentioned in the previous section, the relation $Do(\gamma, s, s')$ has a natural Prolog implementation in terms of the one-step interpreter $trans$:

```
offline(Prog,S0,Sf) :- final(Prog,S0), S0 = Sf ;
                       trans(Prog,S0,Prog1,S1),
                       offline(Prog1,S1,Sf).
```

A Brave On-Line Interpreter

The difference between on- and off-line interpretation of a Golog program is that the former must select a first action from its program, commit to it (or, in the physical world, do it), then repeat with the rest of the program. The following is such an interpreter:

```
online(Prog,S0,Sf) :- final(Prog,S0), S0 = Sf ;
  trans(Prog,S0,Prog1,S1),    /* Select a first action of Prog. */
  ( S1=S0 ; S1=do(A,S0), execute(A) ),
  !,                          /* Commit to this action A. */
  online(Prog1,S1,Sf).
```

The on and off-line interpreters differ mainly in the latter's use of the Prolog cut (!) to prevent backtracking to $trans$ to select an alternative first action of $Prog$.[11] The effect is to

[11] Keep in mind that Golog programs may be nondeterministic.

commit to the first action selected by $trans$. We need this because a robot cannot undo actions that it has actually performed in the physical world. It is this commitment that qualifies the clause to be understood as on-line interpreter. We refer to it as *brave* because it may well reach a dead-end, even if the program it is interpreting has a terminating situation.

A Cautious On-Line Interpreter

To avoid the possibility of following dead-end paths, one can define a *cautious* on-line interpreter as follows:

```
online(Prog,S0,Sf) :- final(Prog,S0), S0 = Sf ;
  trans(Prog,S0,Prog1,S1),  /* Select a first action of Prog. */
  offline(Prog1,S1,S2),     /* Make sure the rest of Prog terminates.*/
  ( S1=S0 ; S1=do(A,S0), execute(A) ),
  !,                        /* Commit to this action A. */
  online(Prog1,S1,Sf).
```

This is much more cautious than its brave counterpart; it commits to a first action only if that action is guaranteed to lead to a successful off-line termination of the program (assuming there are no exogenous actions). Provided this program has a terminating situation, a cautious on-line interpreter never reaches a dead-end.

A cautious on-line interpreter appeals to the off-line execution of the robot's program (in the process of guaranteeing that after committing to a program action, the remainder of the program terminates). Therefore, this requirement precludes cautious interpretation of robot programs that appeal to sensing actions [Levesque, 1996, Golden and Weld, 1996], since such actions cannot be handled by the off-line interpreter.[12] Because the brave interpreter never looks ahead, it is suitable for programs with sense actions. The price it pays for this is a greater risk of following dead-end paths.

Committing to an action is an intrinsically *procedural* notion, and so it is highly desirable, in any logical approach to modeling dynamical systems, to very tightly delimit where in the theory and implementation this nonlogical notion appears. In our case, we can point to the Prolog cut operator in the above on-line interpreters as the exact point at which the procedural notion of commitment is realized.

The full version of the cautious on-line interpreter is enclosed in Appendix A.1.

[12]However, one could imagine a cautious interpreter that verifies off-line that the program terminates for all possible outcomes of its sensing actions.

2.4 Markov Decision Processes

We begin with some basic background on MDPs (see [Ross, 1983, Bertsekas and Tsitsiklis, 1996, Boutilier, 1999, Boutilier *et al.*, 1999, Puterman, 1994] for further details on MDPs). We assume that we have a stochastic discrete time dynamical system to be controlled by some decision maker and in addition we assume that the system can experience the impact of various *exogenous* actions or events (they are beyond the agent's control). A fully observable MDP $M = \langle \mathcal{S}, \mathcal{A}, \Pr, R \rangle$ comprises the following components. $\mathcal{S}$ is a set of *states* of the system being controlled. The state changes over time, possibly in response to the occurrence of exogenous actions (including nature's actions) and/or actions on the part of the decision maker. A set $\mathcal{A}$ is a set of *actions* which influence the system state. Dynamics are given by $\Pr : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$; here $\Pr(s_i, a, s_j)$ denotes the probability that action a, when executed at state s_i, induces a transition to s_j.[13] $R : \mathcal{S} \times \mathcal{A} \to \Re$ (or simply $R : \mathcal{S} \to \Re$) is a real-valued, bounded *reward function*: $R(s)$ is the instantaneous reward an agent receives for entering state s; $R(s, a)$ is the reward for entering s after doing a.[14] The notation $\Pr(s_i, a, s_j)$ emphasizes an important assumption that the next state s_j of the system depends only on the last state s_i and action a and does not depend on any previous states or actions (*Markov property*). This notation emphasizes also another important assumption that probabilities of transitions between states do not change with time, i.e., that an environment is *stationary* and its probabilistic properties remain the same no matter for how long a decision maker is acting in this environment. Similarly, the rewards are functions only of the last state and action. When effects of the exogenous actions are folded into the transition probabilities associated with the agent's action, such models of actions are called *implicit event models*. Alternatively, we can think of transitions as being determined by the effects of the agent's chosen action and those of certain exogenous events beyond the decision maker's control, each of which may occur with a certain probability. When the effects of actions are decomposed in this fashion, the action model is referred as an *explicit event model* [Boutilier, 1999, Boutilier *et al.*, 1999]. In this case, we can assume that exogenous events at different times are independent and have stationary distributions. To simplify mathematical assumptions that have to be made about an MDP we consider only the case when $\mathcal{S}, \mathcal{A}$ are *finite* sets (i.e., actions and exogenous events have only finite number of outcomes). There are monographs that consider cases when state $\mathcal{S}$ and action $\mathcal{A}$ spaces have

[13] In general case, different actions can be available at different states: $\mathcal{A}_s$ denotes the set of feasible actions for state s.

[14] Rewards can be both positive and negative; in the latter case, they can be understood as punishments or costs.

more than finitely many elements (e.g., see [Blackwell, 1965, Dynkin and Yushkevich, 1979, Hinderer, 1970, Bertsekas and Shreve, 1978, Hernández-Lerma and Lasserre, 1996]).

The process is *fully observable* if the agent cannot predict with certainty the state that will be reached when an action is taken, but it can observe that state precisely once it is reached. Another extreme case would be *non-observable* system in which the agent receives no information about the system state during execution. In such open-loop systems, the agent receives no useful feedback about the result of its actions. An intermediate case between these two extreme models is when the agent receives incomplete or noisy information about the system state: *partially observable* MDPs or POMDPs. Formally, POMDP is described as a tuple $M = \langle \mathcal{S}, \mathcal{A}, \Pr, R, \mathcal{O}, Z, b_0 \rangle$, where observation space $\mathcal{O}$ is introduced to model agent's observational capabilities. The agent receives an observation from this set at each stage prior to choosing its action at that stage. The probability distribution $Z = \Pr(o_h | s_i, a_k, s_j)$ determines the probability that the agent observes $o_h \in \mathcal{O}$ given that it performs a_k in state s_i and ends up in state s_j. Note that as with actions, we assume that observational distributions are stationary. The information about the initial state of the world is represented by some belief $b_0(s) = \Pr(s_0 = s)$ that is the probability distribution over all possible states $s \in \mathcal{S}$. In the case of fully observable MDPs, $\mathcal{O} = \mathcal{S}$ and $\Pr(o_h | s_i, a_k, s_j)$=1 iff $o_h = s_j$; otherwise, this probability is 0. In the case of non-observable systems, $\mathcal{O} = \{o\}$: the same observation is reported at each stage, revealing no information about the state.

In subsequent chapters, we assume that an environment is fully observable: this corresponds to a dynamic closed-world assumption that we make in robotics implementations. In Chapter 3, we introduce sensing actions and propose an approach that allows us to implement full observability assumption in our logical framework.

In decision theory, the goal of an agent is to maximize the expected utility (an average total accumulated reward) earned over some time frame. The horizon H defines this time frame by specifying the number of time steps the agent must plan for. The horizon can be finite (if it is a positive integer number) or infinite. In some tasks, MDP has a clearly defined terminal decision epoch, but it can remain unknown what the precise value of horizon is until the decision process will terminate (go into an absorbing state). In these cases (called indefinite horizon problems or episodic tasks), one cannot choose a fixed finite horizon unless a bound needed to solve a problem is known. These tasks can also be conveniently represented as infinite horizon decision processes because the exact number of decision epochs cannot be determined ahead of time. In all cases when the horizon H is infinity, one way to guarantee convergence of the sum of rewards earned at different stages is to multiply rewards by a discount factor γ. A discount factor γ is also used to indicate how rewards earned at different stages should be weighted. In general, the more delayed a reward is, the smaller will be its weight. Therefore, $0 \leq \gamma < 1$ is a real number indicating by how much a reward should be scaled down for every time step delay. A reward earned k steps in the future is scaled down by γ^k. Another way of dealing with the infinite horizon is to consider an average reward per decision epoch.

2.4.1 Optimality Criteria and Decision Algorithms

To choose actions we must follow some *policy*. A policy (a course of action, contingency plan, universal plan, strategy, a closed-loop control policy) specifies the decision rule to be used at each stage. Formally, a policy π is a sequence of decision rules μ_t, i.e., $\pi = (\mu_1, \ldots, \mu_t, \ldots)$. In general, we can place no restrictions on the class of allowable policies, and we can define as a decision rule (a rule for choosing actions) any computable procedure for action selection based on a portion of past history of actions and states. For example, the action specified by a policy can be based on the history of all past actions and the history of observations about the system state up to current point: this history can be denoted h_t. The history $h_t = (s_1, a_1, \ldots, s_{t-1}, a_{t-1}, s_t)$ can be defined recursively as $h_t = (h_{t-1}, a_{t-1}, s_t)$, where s_i and a_i denote the state and action at decision epoch i and h_1 is simply an initial state s_1. A rule for choosing actions also may be randomized in the sense that it chooses action $a \in \mathcal{A}$ with some probability $\mu_t(a, h_t)$ that depends on the history h_t of past actions and observations. In the latter case, $\mu_t(a, h_t) = \mathrm{Pr}_t(a_t = a | s_t, s_{t-1}, a_{t-1}, \ldots, s_1, a_1)$. For example, when in state s, a coin or a die may be tossed to determine which of several possible actions to take. However, the kind of coin (die) used may depend on the previous sequence of states and actions taken. Degenerate probability distributions correspond to deterministic action choice. More specifically, nonrandomized policies π are sequences of mappings from the set of observable histories to actions, that is $\pi = (\mu_1, \ldots, \mu_t, \ldots)$, where each μ_t is a function that maps the history h_t into $\mathcal{A}$. This decision rule μ_t is said to be deterministic because it chooses an action with certainty.

An important subclass of the class of all policies is the class of Markovian (memoryless) policies: if μ_t is parametrized only by the current state s_t, then policy π that consists of a sequence of $\mu_t(s, a) = \mathrm{Pr}_t(a_t = a | s_t = s)$ is a Markovian policy. If, for each t, μ_t is parametrized only by (s_1, s_t), then a policy π is a semi-Markovian policy. In the case of nonrandomized (deterministic) Markovian policies, each μ_t is simply a function that maps a current state s_t into an action $a_t \in \mathcal{A}$ to be chosen by a decision maker.

In the class of nonrandomized Markovian policies, we can distinguish two interesting subclasses: *stationary* (time invariant) policies and other Markovian policies that depend on the current stage. A policy is said to be stationary if the action it chooses at stage n only depends on the state of the process at stage n. A stationary policy has the form $\pi = (\mu, \mu, \ldots)$, where μ is a decision rule used to choose an action at any moment of time. In the case of stationary Markovian policies, it is common to use the same notation π to denote both a policy and a rule μ for choosing actions at any state s. In other words, a stationary policy is a function $\pi : \mathcal{S} \to \mathcal{A}$ mapping the state space $\mathcal{S}$ into the action space $\mathcal{A}$, with the meaning that for each state s, $\pi(s)$ denotes the action the policy chooses when in state s. For an MDP with a finite horizon H, a *nonstationary* policy $\pi : \mathcal{S} \times \{1, \cdots, H\} \to A$ associates with each state s and stage-to-go $n \leq H$ an action $\pi(s, n)$ to be executed at s with n stages remaining. In other words, an agent finding itself in state s at stage n must choose an action $\pi(s, n) = a$ that results stochastically in the next state s'.

In the class of randomized Markovian policies, we can similarly distinguish stationary and nonstationary policies, but in this case they will be probability distributions over $\mathcal{A}$. In a more general setting one can consider also nonmarkovian stationary policies, but we will not consider them in this thesis.

The *decision problem* faced by the agent in an MDP is to adopt a course of action π that maximizes expected reward accumulated by implementing that course of action over some horizon of interest. The *expected value* of policy π depends on the specific objectives. A *finite-horizon* decision problem with horizon H measures the value of π as

$$V^{\pi}(s) = E_{\pi}(\sum_{n=0}^{H} R(s_n) \mid s_0 = s),$$

where E_{π} represents the conditional expectation, given that policy π is employed and the notation $V^{\pi}(s)$ indicates that the *value* of a policy π depends on the state s in which the process begins.

In a discounted infinite horizon MDP,

$$V^{\pi}(s) = E_{\pi}(\sum_{n=0}^{\infty} \gamma^n R(s_n) \mid s_0 = s).$$

There are also other optimality criteria, but we do not consider them in this thesis.

The function V^{π} is called the *state-value function* for policy π. Similarly, one can define the value of taking action a in state s under policy π, denoted $Q^{\pi}(s, a)$, as the expected return that can be accumulated if a decision maker takes the action a in s and thereafter follows policy π [Watkins, 1989, Watkins and Dayan, 1992]:

$$Q^{\pi}(s,a) = E_{\pi}(\sum_{n=0}^{\infty} \gamma^n R(s_n) \mid s_0 = s, a_0 = a),$$

the function $Q^{\pi}(s,a)$ is called the *action-value function* for policy π.

A policy π^* is *optimal* if, for all $s \in \mathcal{S}$ and all policies π, we have $V^{\pi^*}(s) \geq V^{\pi}(s)$. There is always at least one policy that is better than or equal to all other policies. Optimal policies share the same state-value function, called the *optimal state-value function*, denoted V^*, and defined as $V^*(s) = max_{\pi} V^{\pi}(s)$, for all $s \in \mathcal{S}$ (in other words, V^* exists and is unique). Note that policies we consider are Markovian (the choice of action does not depend on history of past states and actions). It can be shown that optimal courses of action lie within these classes of policies: nonstationary Markovian policies for finite-horizon problems [Bellman, 1957, Derman, 1970], and stationary deterministic Markovian policies for infinite-horizon problems [Bellman, 1957, Howard, 1960, Blackwell, 1962, Howard, 1971, Ross, 1983].

Consider the problem of finding an optimal policy for some fixed, finite horizon H. A simple algorithm for constructing optimal policies is *value iteration* [Bellman, 1957, Howard, 1971, Puterman, 1994]. Define the n-stage-to-go value function V_n by setting $V_0(s_i) = R(s_i)$ and, for all $1 \leq n \leq H$:

$$V_n(s_i) = R(s_i) + \max_{a \in \mathcal{A}} \{ \sum_{s_j \in \mathcal{S}} \Pr(s_i, a, s_j) V_{n-1}(s_j) \} \tag{2.25}$$

The computation of $V_n(s_i)$ given $V_{n-1}(s_j)$ is known as a Bellman backup and the equation (2.25) is called the Bellman optimality equation. Using this equation, one can compute in sequence the optimal state value functions up to the horizon H of interest. Once this sequence of the value functions is computed, one can easily find an optimal policy. By setting $\pi(s_i, n)$ to the action a maximizing the right-hand term, the resulting policy π will be optimal. Indeed, an *optimal* policy is one with maximum expected value at each state-stage pair. In general case, an optimal policy is nonstationary because of finite horizon. This remarkable algorithm is based on "The Principle of Optimality" verbally stated in [Bellman, 1957], p.83:

> "An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

This value iteration algorithm takes H iterations. At each iteration, it does $|\mathcal{A}|$ computations of $|\mathcal{S}| \times |\mathcal{S}|$ matrix times $|\mathcal{S}|$-vector. Thus, in total it requires $O(H \times |\mathcal{A}| \times |\mathcal{S}|^3)$ operations.

Because the number of states grows exponentially with increase in the number of features used to represent each state (Bellman's "curse of dimensionality"), the value iteration algorithm becomes impractical once the number of boolean (propositional) features becomes large (e.g., state of the art MDP solvers can compute optimal policies in cases when the number of features is no more than 35, i.e., when the number of states has the order of $2^{35} > 3 \cdot 10^{10}$ states). Different compact representations of state space and approximation techniques are reviewed in [Boutilier *et al.*, 1999, Bertsekas and Tsitsiklis, 1996].

Consider the problem of finding an optimal policy in a discounted infinite horizon MDP. Because optimal policies in this case are stationary Markovian policies, it can be shown that the conditional expectation E_π in the optimality criterion in this case can be rewritten as the following recurrences (*Bellman equations*):

$$V^\pi(s_i) = R(s_i) + \gamma \sum_{s_j \in \mathcal{S}} \Pr(s_i, \pi(s_i), s_j) V^\pi(s_j) \tag{2.26}$$

For each given policy π, this is a system of $|\mathcal{S}|$ linear equations with $|\mathcal{S}|$ unknown variables $V^\pi(s_i)$ (each variable is the value of state s_i for policy π). Each equation states an intuitively clear fact that the value of state s_i under policy π is the sum of the immediate reward received in s_i and discounted expected value of the successor state. Since there exists a unique, optimal value function, V^*, [Blackwell, 1962] for any optimal policy π^* we have

$$V^*(s_i) = R(s_i) + \gamma \sum_{s_j \in \mathcal{S}} \Pr(s_i, \pi^*(s_i), s_j) V^*(s_j).$$

Hence, this system of *Bellman optimality equations* can be written in a special form without reference to any specific policy

$$V^*(s_i) = \max_{a \in \mathcal{A}} \{R(s_i) + \gamma \sum_{s_j \in \mathcal{S}} \Pr(s_i, a, s_j) V^*(s_j)\}, \tag{2.27}$$

because the action $a = \pi^*(s_i)$ chosen by an optimal policy must be the best action in every state. A policy that prescribes to each state an action that maximizes the sum of the immediate payoff and the discounted expected value of the successor state is called a *greedy* policy with respect to a value function. If a current estimate (or approximation) of the state value function is considered instead of the actual state value function, then a greedy policy with respect to this estimate can be different from a greedy policy with respect to the actual function. Any policy that is greedy with respect to the optimal value function is an optimal policy.

Equations (2.26) and (2.27) can be conveniently written using vector notation. Let $R = (R(s_1), \ldots, R(s_{|\mathcal{S}|}))^T$ be the vector of payoffs, $V \in \mathcal{R}^{|S|}$ be any vector of $|S|$ real numbers,

$V^\pi = (V^\pi(s_1), \ldots, V^\pi(s_{|\mathcal{S}|}))^T$ be vector of unknown state values under policy π and let $P[\pi]$ be the transition probability matrix under policy π with elements $p_{i,j} = \Pr(s_i, \pi(s_i), s_j)$ for any $1 \leq i, j \leq |\mathcal{S}|$. It is convenient to introduce also two operators in a vector space. Let the linear backup operator under policy π be $B_\pi V = R + \gamma P[\pi] V$ and the nonlinear (Bellman) backup operator (it has no policy subscript) be $BV = max_{\pi \in \mathcal{P}}(R + \gamma P[\pi] V)$, where $\mathcal{P}$ is the set of stationary Markovian policies (it has $|\mathcal{A}|^{|\mathcal{S}|}$ elements). Then, the system of linear equations (2.26) can be written in vector form:

$$V^\pi \;=\; R + \;\gamma P[\pi] V^\pi,$$

or in operator form:

$$V^\pi = B_\pi V^\pi.$$

The policy backup operator B_π is monotone (i.e., for any $|\mathcal{S}|$-dimensional real vectors V_1 and V_2 such that $V_1 \leq V_2$ and for any stationary Markovian policy π we have $B_\pi V_1 \leq B_\pi V_2$). Let $\| V \| = max_{x \in \mathcal{S}} |V(x)|$. The operator B_π is a contraction operator in maximum norm, i.e., for any $V_1, V_2 \in \mathcal{R}^{|S|}$

$$\| B_\pi V_1 - B_\pi V_2 \| \leq \gamma \| V_1 - V_2 \| .$$

Therefore, by the contraction mapping theorem [Kolmogorov and Fomin, 1970], V^π, the value function of π, is the unique fixed point of B^π. Solving this allows to evaluate a fixed stationary policy π. Policy evaluation can be achieved by any algorithm that can solve linear systems of equations. The system (2.27) in operator form:

$$V^* = BV^*.$$

The Bellman operator B is also monotone: if $V_1(s) \leq V_2(s)$, then $BV_1 \leq BV_2$; this is also a contraction operator because $\| BV - V^* \| \leq \gamma \| V - V^* \|$ for any $V \in \mathcal{R}^{|S|}$. The optimal value function V^* is the unique fixed point of the Bellman backup operator B. Because B is nonlinear operator solving this in a reasonable amount of time requires clever search in the set $\mathcal{P}$ (recall that $\mathcal{P}$ is the set of stationary Markovian policies introduced above). Let B_π^k be the composition of k copies of B_π and B^k be interpreted similarly. Then we can say that for any $V \in \mathcal{R}^{|S|}$

$$V^\pi = \lim_{k \to \infty} B_\pi^k V \quad \text{and} \quad V^* = \lim_{k \to \infty} B^k V.$$

There are two most well known dynamic programming algorithms that compute an optimal policy for an MDP with an infinite horizon: policy iteration and value iteration [Puterman, 1994, Bertsekas and Tsitsiklis, 1989, Kaelbling *et al.*, 1996, Bertsekas and Tsitsiklis, 1996].

Policy iteration does search in the set $\mathcal{P}$: begins with an arbitrary policy π_0, then iteratively computes an improved policy π_{n+1} upon the old one π_n ($n \geq 0$). Each iteration has two steps, *policy evaluation* (prediction problem) and *policy improvement* (control problem).

1. At the policy evaluation step, we solve (2.26) by computing the value function $V^{\pi_n}(s)$ for each $s \in \mathcal{S}$. This can be achieved using the method of successive approximations if the model of the environment is available (we know a priori the transition probabilities P and rewards R). This model-based method has two main variations. The (Jacobi) synchronous successive approximation method updates the values for all states simultaneously: $V_{i+1} = B_{\pi_n} V_i$, where V_i is an ith approximation to solution V^{π_n} of (2.26). This synchronous algorithm converges to V^{π_n} because B_{π_n} is a contraction operator. The (Gauss-Seidel) asynchronous approximation method updates the values for the states as follows. At each step, some subset $G \subseteq \mathcal{S}$ is chosen and for each $s \in G$, $V_{i+1}(s) = B_{\pi_n}(s)V_i(s)$, and $V_{i+1}(s) = V_i(s)$ if $s \notin G$, where the linear backup operator for state x is
$$B_{\pi_n}(x)V = R(x) + \gamma \sum_{y \in \mathcal{S}} \Pr(x, \pi_n(x), y)V(y).$$
The strategy for selecting G at each stage must ensure that every state s is updated often; convergence is proved in [Bertsekas and Tsitsiklis, 1989, Bertsekas and Tsitsiklis, 1996].

2. Once policy is evaluated, we can improve the policy π_n, i.e., for each $x \in \mathcal{S}$, find an action a^* which maximizes
$$Q^{\pi_n}_{n+1}(x, a) = R(x) + \gamma \sum_{y \in \mathcal{S}} \Pr(x, a, y)V^{\pi_n}(y).$$
If $Q^{\pi_n}_{n+1}(x, a^*) > V^{\pi_n}(x)$, then a new policy $\pi_{n+1}(x) = a^*$; otherwise, $\pi_{n+1}(x) = \pi_n(x)$.

When we discussed the policy evaluation step above, we mentioned model-based methods. Besides them, there are also model-free methods that can be used to evaluate a policy π_n even if the model of the environment is not available (i.e., they can be used *on-line* when the agent interacts with an unknown environment): Monte-Carlo methods [Barto and Duff, 1994, Tsitsiklis, 2002] and the method TD(0) [Sutton, 1988, Sutton, 1995, Sutton and Barto, 1998a]. When the agent executes (or simulates execution of) an action $\pi_n(s)$, observes a transition from state s to state s' and receives reward r, TD(0) adjusts $V(s)$ as follows:
$$V_i(s) \leftarrow V_{i-1}(s) + \alpha_i(s)[r + \gamma V_{i-1}(s') - V_{i-1}(s)],$$

where $0 \leq \alpha_i(s) \leq 1$ is a learning rate, that depends on state s, and $r+\gamma V_{i-1}(s')$ is the sampled estimate of right-hand-side of the Bellman equation. For $u \neq s$,

$$V_i(u) \leftarrow V_{i-1}(u).$$

In [Singh, 1993, Jaakkola *et al.*, 1994], it is proved that with probability 1, V_i converges to V^{π_n} under the usual assumptions that guarantee convergence of stochastic approximation algorithms [Robbins and Monro, 1951, Benveniste *et al.*, 1990], i.e., if every state is updated often and for any $s \in \mathcal{S}$ the sequence $\alpha_i(s)$ is decayed at a suitable rate: $\sum_{i=0}^{\infty} \alpha_i(s) = \infty$ and $\sum_{i=0}^{\infty} \alpha_i^2(s) < \infty$.

The policy iteration algorithm continues until $\pi_{N+1}(s) = \pi_N(s)$ at some iteration N for every state $s \in \mathcal{S}$. Because the set $\mathcal{P}$ of stationary Markovian policies is finite, this algorithm converges to an optimal policy since each new policy is guaranteed to be a strict improvement over previous one (unless it is already optimal).

As we see, in policy iteration, each iteration involves policy evaluation step which may itself require a time consuming computation in the state space. *Value iteration algorithm* avoids this by replacing policy evaluation step by a single backup of each state. It starts with an arbitrary value function V_0, but instead of applying B_π for a particular policy, it applies the Bellman operator B, which maximizes over all actions:

$$V_{n+1} \leftarrow BV_n.$$

This is a (Jacobi) synchronous value iteration algorithm; convergence can be proved because B is a contraction operator. Similarly to policy evaluation, there is asynchronous value iteration when not all states are updated at once. At each step, some $G \subseteq \mathcal{S}$ is chosen and for each $s \in G$, $V_{n+1}(s) = B(s)V_n(s)$; otherwise $V_{n+1}(s) = V_n(s)$ (for $x \notin \mathcal{S}$), where the Bellman backup operator for state x is

$$B(x)V = R(x) + \gamma \max_{a \in \mathcal{A}} \{\sum_{y \in \mathcal{S}} \Pr(x, a, y)V(y)\}.$$

The asynchronous value iteration algorithm allows the agent to sample the state space by randomly selecting the state to which the update equation is applied. Asynchronous algorithm includes synchronous and Gauss-Seidel algorithms as special cases: if for each step $G = \mathcal{S}$, then this is a synchronous dynamic programming; Gauss-Seidel results when each set G consists of a single state and the collection of states is an ordered sequence of states (from the state 1 up to the state $|\mathcal{S}|$ and then again starting from the state 1, etc.). Convergence is guaranteed if the strategy for selecting states for backup never eliminates any state from possible selection in the future.

As for an effective stopping criterion for the value iteration algorithm, it can be formulated in terms of the Bellman residual of the current value function [Williams and Baird, 1993b, Singh and Yee, 1994]. More specifically, if $\| V_{n+1} - V_n \|$, the maximum difference between two successive value functions, is less than ϵ, then the value function $V^{\pi_{n+1}}$ of the greedy policy π_{n+1} with respect to V_n, differs from the value function V^* of the optimal policy by no more than $\frac{2\epsilon\gamma}{(1-\gamma)}$ at any state. As for computational complexity of dynamic programming algorithms, each iteration of the synchronous value iteration algorithm can be performed in $O(|\mathcal{A}||\mathcal{S}|^2)$ steps, or faster if there is sparsity in the transition function. However, the number of iterations required can grow exponentially as the discount factor approaches 1. In practice, policy iteration is known to converge in fewer iterations than value iteration, although the per-iteration costs of $O(|\mathcal{A}||\mathcal{S}|^2 + |S|^3)$ can be prohibitive in the case of large state spaces. Additional details about dynamic programming algorithms, about their variations and about connections with the field of reinforcement learning can be found in [Puterman, 1994, Bertsekas and Tsitsiklis, 1996, Kaelbling *et al.*, 1996].

MDPs are natural models for decision-theoretic planning, but traditional framework requires explicit state and action enumeration, which grow exponentially with the number of domain features. On the other hand, most AI problems involving decision-theoretic planning can be viewed in terms of logical propositions, random variables, relations between objects, etc. Logical representations have significant advantage over traditional state-based representations because they can allow compact specifications of the system. For this reason, many researchers studied how to represent and solve large state space MDPs using alternative representation techniques, most notably probabilistic STRIPS [Hanks and McDermott, 1994, Kushmerick *et al.*, 1995], dynamic Bayesian networks [Dean and Kanazawa, 1989, Boutilier and Goldszmidt, decision trees [Dearden and Boutilier, 1997, Boutilier *et al.*, 2000b], logical rules [Poole, 1997, Poole, 1998], algebraic decision diagrams [Hoey *et al.*, 1999, Hoey *et al.*, 2000] and the situation calculus [Boutilier *et al.*, 2000a, Boutilier *et al.*, 2001]. Most of these representations and related developments of efficient methods for solving MDPs are reviewed in [Boutilier *et al.*, 1999]. In Chapter 5 we consider our approach to representation of MDPs in the predicate logic.

2.4.2 Decision Tree Search-Based Methods

When the system is known to start in a given state s_0, the reachability structure of the MDP can be exploited, restricting value and policy computations to states reachable by some sequence of actions from s_0. This form of *directed value iteration* can be effected by building a search tree rooted at state s_0: its successors at level 1 of the tree are possible actions; the successors at level 2 of any action node are those states that can be reached with nonzero probability when that action is taken at state s_0; and deeper levels of the tree are defined recursively in the same way. For an MDP with finite horizon k, the tree is built to level $2k$: the value of any state is given by the maximum among all values of its successor actions, and the value of an action is given by the expected value of its successor states.[15] It is easy to see that the value $V(s_0)$ of the state s_0 at the root of a tree with $2n$ levels is precisely $V_n(s_0)$ defined in the equation (2.25) for value iteration. This observation provides the basis for the well known relationships between heuristic search techniques and other dynamic programming algorithms. For short horizon problems with small branching factor, the advantage of tree search over other algorithms becomes apparent from the fact that only states reachable from the initial state s_0 need to be taken into account to compute the value function (all other states may remain unexplored). Search-based approaches to solving MDPs can use heuristics, learning, sampling and pruning to improve their efficiency [Barto *et al.*, 1995, Dearden and Boutilier, 1997, Kearns *et al.*, 1999, Koenig and Simmons, 1995]. Declarative search control knowledge, used successfully in classical planning [Bacchus and Kabanza, 1996, Bacchus and Kabanza, 2000, Gabaldon, 2003], might also be used to prune the search space. In an MDP, this could be viewed as restricting the set of policies considered. This type of approach has been explored in the more general context of value iteration for MDPs by Parr and Russell [Parr and Russell, 1998]: they use a finite state machine to model a partial policy and devise an algorithm to find the optimal policy consistent with the constraints imposed by the FSM. Their work as well as related work will be discussed later in Chapter 5.

Historically, the idea of interrupting computationally intensive search for a plan earlier in favor of executing (possibly suboptimal, but promising) actions from an initial segment that can be extended later to a plan leading to a goal was first formulated in [Korf, 1990]. In particular, Korf presents in this paper a new algorithm, called Real-Time-A^* (*RTA**), that makes locally optimal decisions and is guaranteed to find a solution in a finite problem space with positive edge costs and finite heuristic values if a goal state is reachable from every state. Simulations

[15]States at the leaves are assigned their reward value.

on several sliding tile puzzles (with a well known Manhattan Distance heuristic function) show this algorithm can outperform (in terms of total running time) other algorithms that try to compute a plan by looking ahead up to a goal state before committing even to a very first action from the plan. However, *RTA** is a suboptimal algorithm in a sense that it can find plans with a length greater than optimal. A learning version of the same algorithm, called Learning Real-Time-A^* (*LRTA**), improves its performance over successive problem solving trials by learning more accurate heuristic values over repeated problem solving trials. This learning version (when search horizon is 1) can be summarized as follows, assuming that $f(s)$ is a heuristic function for a distance to a goal (i.e., for a minimal number of actions that have to be executed starting from s to reach one of a goal states) and $successor(s, a)$ is a state reachable from s by executing action $a \in \mathcal{A}(s)$. Initially, values of all states $s \in \mathcal{S}$ are $V(s) = f(s)$. The algorithms starts in state s_0 and proceeds iteratively:

1. $s \leftarrow$ the current state.
2. If s is a goal state, then stop successfully.
3. Otherwise, compute $a = arg\min_{a \in \mathcal{A}(s)} V(successor(s, a))$.
4. Update $V(s) \leftarrow max(V(s), 1 + V(successor(s, a)))$.
5. Execute action a in reality (or simulate this execution on computer); this leads to a state $successor(s, a)$.
6. Repeat.

Thus, *LRTA** looks one action execution ahead and always greedily chooses an action that leads to a successor state with the minimal value (ties can be broken randomly). The planning time of *LRTA** between action executions is linear in $|\mathcal{A}(s)|$, the number of actions available in the current state. Therefore, if $|\mathcal{A}(s)|$ is bounded (or does not depend on the number of states), then the planning time between action executions is constant (or does not depend on $|\mathcal{S}|$, respectively). *LRTA** is extensively analyzed in [Ishida, 1997].

Both algorithms, *RTA** and *LRTA**, were proposed for deterministic domains with complete information, where each state has only one successor. However, *real-time heuristic search methods* (this term is suggested by Korf) that interleave search with action execution can be also successfully applied to other domains. In particular, [Genesereth and Nourbakhsh, 1993, Nourbakhsh, 1997] present pruning rules for problem solving with incomplete information.

These rules are domain-independent and lead to savings in planning costs. The rules are of special importance in the case of interleaved planning and execution in that they allow the planner to terminate search without planning to the goal. The min-max *LRTA** algorithm, a simple extension of *LRTA** to non-deterministic domains is proposed and analyzed in [Koenig and Simmons, 1995, Koenig, 2001]. The authors describe which non-deterministic domains can be solved using their new algorithm, and analyze its performance for these domains. The min-max *LRTA** algorithm (with a search horizon 1) is summarized in [Koenig and Simmons, 1995 as follows. Note that in nondeterministic domains actions can lead to several different states: $successor(s, a) \subseteq \mathcal{S}$. Initially, values of all states $s \in \mathcal{S}$ are $V(s) = f(s)$ (a heuristic function $f(s)$ is domain dependent). The algorithm starts in state s_0 and proceeds iteratively:

1. $s \leftarrow$ the current state.
2. If s is a goal state, then stop successfully.
3. $$a = arg \min_{a \in \mathcal{A}(s)} \max_{s' \in successor(s,a)} V(s').$$
4. $$V(s) \leftarrow max(V(s), 1 + \max_{s' \in successor(s,a)} V(s')).$$
5. Execute action a in reality; nature selects the new state from the set $successor(s, a) \subseteq \mathcal{S}$.
6. Repeat.

In contrast to traditional (off-line) search techniques, which must plan for every possible contingency in an environment (where nature, and possibly other agents, can execute actions interfering with the agent's actions), real-time search methods only need to choose actions for those outcomes that actually occur. In a complex large-scale domain, without interleaving planning and execution, the agent has to compute a potentially large conditional plan. But with interleaving, the agent has to find only the beginning of such a plan. Because the agent repeats the process of planning and executing from the state that actually resulted from the execution of the initial subplan, only this state matters. The purely planning agent must consider all states that could have resulted from execution of this subplan. Thus, real-time search methods can potentially decrease search time, although possibly at the expense of computing longer suboptimal plans.

An algorithm that combines ideas of a forward look-ahead search from an initial state s_0 with asynchronous value iteration algorithm is called Real-Time Dynamic Programming (*RTDP*). It is introduced and analyzed in [Barto *et al.*, 1995, Bradtke, 1994]. From the dynamic programming perspective, *LRTA** can be characterized as deterministic specialization of the asynchronous value iteration algorithm applied on-line. From the control systems perspective, *LRTA** is a kind of *receding horizon control*[16] that can accommodate the possibility of closed-loop control and can improve over time because it accumulates the results of shallow searches forward by updating the evaluation function. *RTDP* extends *LRTA** in two ways: it generalizes *LRTA** to stochastic problems, and it includes the option of backing up values of many states in the time intervals between the executions of actions. Both *RTDP* and *LRTA** may be augmented by look-ahead search. This means that instead of using values of successors of the current state, each of this algorithms can perform an off-line forward search from the current state to a depth determined by the amount of computational resources available. *RTDP* refers to cases when decision-theoretic planning is interleaved with executing of actions as follows. First, an action selected for execution is the greedy action with respect to the most recent estimate of the state value function (ties can be resolved randomly). Second, between the execution of actions, the value of the current state (and possibly also any other states generated by look-ahead search) is backed up using the current estimates of values of successor states. In the discounted case, the only condition that is required for convergence of the associated asynchronous value iteration algorithm to the optimal value function is that no state is completely ruled out from backing up. This can be implemented in the trial-based *RTDP* by choosing a new start state randomly in the state space each time when a new trial must be started again after reaching one of goal states. However, it is possible to focus *RTDP* to start each trial only in one of designated start states S_0 and consider only those states s' (called *relevant* in [Barto *et al.*, 1995]) that can be reached from $s_0 \in S_0$ by following an optimal policy. Conditions under which this is possible are stated precisely in [Barto *et al.*, 1995, Bradtke, 1994]. Further developments of *RTDP* are studied in [Bonet and Geffner, 2003, Feng *et al.*, 2003].

Another method for efficient planning in MDPs that focuses on a restricted set of states is considered in [Dean *et al.*, 1995]. To cope with large state spaces, a highly restricted subset of the entire state space, called *envelope*, is proposed as a domain for computing a partial policy. Their basic algorithm consists of two stages: envelope alteration followed by policy generation.

[16]The receding horizon approach to MDPs is studied in [Hernández-Lerma and Lasserre, 1990, Chang and Marcus, 2003].

More specifically, the algorithm takes an envelope and a policy as input and generates as output a new envelope and policy. The algorithm is applied in the manner of iterative refinement (with more than one invocation of the algorithm). In this paper, the authors consider the meta-level control problem of *deliberation scheduling* related to allocating computational resources to several iterative refinement routines. They propose *precursor* models and *recurrent* models of planning and execution. In precursor models decision making is performed prior to execution, but in recurrent models decision making is performed in parallel with execution, accounting for actually observed states. Using the robot navigation domain (with about 1000 states) as a testbed, the paper provides an experimental comparison of *RTDP*, a version of recurrent deliberation algorithm, and two algorithms based on the standard AI planning assumptions: trajectory planning with replanning (an initial path to the goal is found, if the agent falls off the path, the reflexes are executed until a new path is found) and trajectory planning with recover (if the agent falls off the nominal path, a path is planned back to the original path, rather than to the goal).

The idea of interleaving decision-theoretic planning with execution realized in *RTDP* suggests a possible way of circumventing computational difficulties associated with decision tree search algorithms (in *decision analysis*, see [Raiffa, 1968], decision trees are called also *decision-flow diagrams*). These difficulties are inherent in many practical problems, where the branching factor of a decision tree is large and a deep decision tree must be constructed (if a finite horizon is a large number, or if infinite horizon problem must be approximated by a finite depth decision tree). More specifically, the agent can build a look-ahead tree up to some depth (depth can depend on time pressures or availability of computational resources). To choose an action optimal with respect to this partial look-ahead tree, the agent can apply the roll-back procedure that computes first values at the leaves of the tree and then determines recursively values at more shallow levels.[17] Then the agent can execute this action and proceed iteratively with building a new look-ahead tree. This offers a significant computational savings because the actual outcome of the execution of the action can be used to prune all those branches which grow from states corresponding to unrealized outcomes: only subtree rooted at the realized state matters. This approach is proposed and analyzed in [Dearden and Boutilier, 1994], which relies in turn on an extension of the alpha-beta tree pruning strategy to trees with 'probability' nodes

[17] [Raiffa, 1968] describes this method as follows. "We first transported ourselves in conceptual time out to the very tips of the tree, where the evaluations are given directly in terms of the data of the problem. We then worked out way backwards by successive use of two devices: 1) an averaging-out process at each chance juncture, and 2) a choice process that selects the path yielding the maximum future evaluation at each decision juncture. We call this the *averaging out and folding back* procedure."

[Ballard, 1983]. While this approach reduces the computational cost of planning in stochastic domains, it sacrifices optimality because searching is done to a fixed depth and a heuristic function is used to estimate the value of states. The paper [Dearden and Boutilier, 1994] considers several new techniques (similar to alpha-beta) for pruning the search tree. Because knowledge of the heuristic function is required by these techniques, the paper also describes how it can be computed using the approach in [Boutilier and Dearden, 1994] (a more recent version is provided in [Dearden and Boutilier, 1997]).

2.5 Robotics

The frameworks described in the subsequent chapters have been implemented on the mobile robot Golem: it is a B21 robot manufactured by Real World Interface (RWI).

Golem is equipped with a laser range finder (it determines distances to surrounding obstacles with a high accuracy) that is installed in a space between base and enclosure. The robot has also 24 sonar proximity sensors, bumpers, one on-board computer with a dual Pentium II 300MHz processor system and 128Mb RAM, and a laptop computer (installed on the console) with one Pentium 200MHz processor and 32Mb RAM (both computers run Linux). The laptop computer has a sound card and can reproduce voice recordings. Golem can communicate with external computers via a radio Ethernet link, but in our implementation we do not use it, because we run all software on-board.

The low-level software that we run on our robot was initially developed in the University of Bonn and Carnegie–Mellon University to control RHINO, another RWI B21 robot; see [Burgard *et al.*, 1998] for details. We mention briefly what modules we run on the on-board computer:

- `tcxServer` – a communication manager that provides asynchronous communication and coordination between the other modules;
- `baseServer` – a program which controls physical motion and gathers information from wheel encoders about the distance traveled by robot;
- `plan` – a path planning module that computes a shortest trajectory from a current location to a goal location using a dynamic programming algorithm;
- `laserServer` – an auxiliary module that determines distances to obstacles using the laser range finder;

- `colliServer` – collision avoidance program that changes the speed and direction of motion whenever the robot's sensors detect obstacles ahead (this program is highly reactive and dynamically modifies the motion of the robot);
- `localize` – a program which determines the probability of the current location given a map and a stream of current measurements from sensors (the map of our office environment is built only once using a neural-networks based sensor interpretation program).

It is important to emphasize that all sensor readings performed by the aforementioned modules are not accessible to the high-level control.

Our Prolog code (we run Eclipse Prolog on the robot's computer) communicates with the low-level software using the software package HLI [Hähnel, 1998, Hähnel *et al.*, 1998] that provides a high-level interface to all other modules. The implementation of HLI relies on the Prolog–C interface of Eclipse Prolog. In particular, to execute the primitive action $startGo$ we call the predicate hli_go_path from Prolog (Golem simply says that it has reached a destination when it has to execute the action $endGo$). The implementation of hli_go_path in HLI ensures that the robot will successfully reach its destination by sending commands to low-level programs even if some of them malfunction [Hähnel *et al.*, 1998]. Because Golem does not have an arm, it vocalizes appropriate audio recordings whenever it has to execute the actions $giveCoffee$ and $pickupCoffee$ and expects an appropriate human reaction.

Chapter 3

The projection task with sensing actions

Sensing and its effects on the knowledge of a robot have been traditionally approached from different perspectives in robotics and logic-based AI. Nevertheless, with the advance of cognitive robotics, there emerged a new trend to designing logical specifications of high level controllers that can be directly implemented on autonomous robotics systems. In particular, [De Giacomo and Levesque, 1999a, Pirri and Finzi, 1999] proposed different logical formalizations of reasoning about physical and sense actions in the situation calculus. These proposals led to successful implementation of robotics control systems [Piazzolla *et al.*, 2000] and [Levesque and Pagnucco, 2000] capable of updating their model of the world by information gathered from sensors.

In this chapter, we propose a new alternative representation of sense actions[1] in the situation calculus that does not rely on the epistemic fluent $K(s', s)$ and accounts for sense actions directly in successor state axioms. In addition, we consider an approach (introduced in [Reiter, 2001b]) to reasoning about effects of actions, knowledge, and sensing that reduces reasoning about knowledge to provability. Finally, we provide a formal comparison of the latter approach with ours. It is shown that under certain assumptions these two approaches lead to essentially equivalent solutions of the forward projection task. In particular, if information delivered by a stream of sense actions is consistent, then, under certain assumptions, in a situation that results from doing all these sense actions (and possibly some other physical actions), a given domain theory (augmented with sensory data, according to the proposal of [Reiter, 2001b]) entails that an agent knows a logical formula ϕ if and only if the domain theory of this agent (with successor state axioms modified according with our approach) en-

[1]The papers [Scherl and Levesque, 1993, Scherl and Levesque, 2003] use the term 'knowledge-producing actions', but we follow [Reiter, 2001a] and call them either 'sense' or 'sensing' actions.

tails ϕ. Later in this chapter, we will see that this comparison provides a formal justification for our approach and also indicates that it remains useful in cases when sense actions gather inconsistent information. For this reason, our approach to dealing with sense actions that relies on modification of successor state axioms can be instrumental for the development of an adequate execution monitoring framework in Chapter 4 because it lifts an unrealistic assumption about observability of all exogenous actions (this assumption was essential in the earlier proposal of [De Giacomo *et al.*, 1998]). We will see also in Chapter 6 that our technical machinery proves to be applicable to the design of an on-line decision-theoretic Golog interpreter that computes an optimal policy to control a mobile robot and needs sensing to determine an outcome of stochastic actions. Later in this chapter, in Section 3.6, we compare our approach with [Demolombe and Pozos-Parra, 2000], who independently from [Soutchanski, 2000], use both physical and sense actions in successor state axioms to describe effects of actions on 'subjective' fluents, but their sense actions are different from sense actions considered here. Their approach is thoroughly investigated and compared with the $K(s, s')$-fluent based approach in the recent paper [Petrick and Levesque, 2002] (see Section 3.6).

The content of this chapter is a significantly revised and extended version of our paper [Soutchanski, 2001a] that develops ideas announced in [Soutchanski, 2000, Soutchanski, 2001b].

3.1 Introduction

In Section 2.1.1, we considered the syntactic form of the successor-state axioms that provide the solution of the frame problem: what logical properties of the world change and what properties persist when an agent executed a *physical* action that intuitively effects only a limited number of properties in the world. Furthermore, in Section 2.1.3 we have seen that this solution can be extended to reasoning about agents' knowledge and *sense* actions by elaborating an earlier proposal by [Moore, 1985] that an *accessibility relation between possible worlds* [Kripke, 1963a, Kripke, 1963b, Hintikka, 1962] can be represented in a situation calculus logical theory by a binary predicate $K(s, s')$ relating two 'epistemically alternative' situations s and s'; this predicate is called *epistemic fluent*. The successor state axiom for this fluent (2.17) (proposed in [Scherl and Levesque, 1993]) characterizes how knowledge changes from the current situation to a successor situation resulting from the execution of a sense action in the current situation. In particular, this axiom (2.17) states that only sense actions may effect the accessibility between situations, more specifically, they 'shrink' the accessibility relation by eliminating those situations which do not 'agree' with the results of sensing. Because fewer possible worlds become accessible after doing sense actions, the knowledge of an agent becomes more complete; in other words, sense actions increase (or produce) knowledge for the agent. Observe that in the axiom (2.17) sense actions are understood in very general terms as actions determining a truth value of an arbitrary (situation suppressed) expression. In addition, the initial theory with knowledge $\mathcal{D}^K_{S_0}$ in general case may include arbitrary first-order sentences about the world itself, about agents knowledge, whether an agent knows what another agent knows, etc.

The generality of this approach to reasoning about knowledge and sensing may hinder practical computer implementations. In particular, realistic high-level controllers implemented in Golog need an account of sensing that is promising computationally. In robotics and other application domains alternative (approximate or considering special cases) approaches to reasoning about sense actions may facilitate implementations and still be sound with respect to the original K-based general specification. For example, in computer animation, [Funge, 1998] incorporates interval arithmetic into the situation calculus to provide a representation of epistemic uncertainty of agents and of the dynamics of this uncertainty when agents obtain a new sensory input. As another example, reduction of knowledge to provability in [Reiter, 2001a, Reiter, 2001b] serves as a sound foundation of knowledge-based programming in an epistemic extension of Golog. In contrast to the version of Golog considered in Section 2.3, programs written in his extension of Golog can include not only physical actions, but also sense actions, and test expressions can mention **Knows**(ϕ) and **KWhether**(ψ) when it is necessary to evaluate what an agent executing the program knows about the world at the current step of execution. Reduction of knowledge to provability is achieved by considering a certain important special case of $\mathcal{D}^K_{S_0}$: when this theory consists exclusively of sentences declaring what the agent knows about the world it inhabits (sentences declaring what is actually true of the world, and what the agent knows about what it knows are not allowed). Because we also intend to consider Golog programs with sensing actions, and we are interested in implementing these programs on a mobile robot, let's formulate several criteria for a logical account of sensing in the situation calculus (some of these criteria arise naturally in the robotics context).

- Sensing actions have to accommodate both binary and real valued data returned from sensors.

- Certain properties in the world vary unpredictably in time and cannot be modeled (e.g., the temperature outdoors, the humidity, the number of people we meet today, etc). Certain other properties can be effected by external agents that function independently and whose actions cannot always be observed (e.g., stock prices). Sequential sense actions have to allow updating of the currently available information about all types of properties (including properties that we mention), but updates should not lead to inconsistencies.

- Sensing needs to be performed whenever it is necessary, in particular, if the information about the world is incomplete and can be enriched only from sensing or if we have good reasons to believe that previous sensory data are no longer valid (e.g., because of complications mentioned in the previous paragraph).

- There has to exist a computationally efficient mechanism of using both sensing information and the information about the initial situation given the specifications of how a dynamical system evolves from one situation to another. Because regression can be an efficient (under certain conditions[2]) mechanism for the solution of the forward projection task, it is desirable to integrate regression with reasoning about sensory information.

- Ideally, the specification of sense actions in the situation calculus has to lead directly to a natural and sound implementation, e.g., in Prolog.

Below we propose a representation of sensing that satisfies these criteria. In addition, we consider a correspondence between our account of sensing and an approach of [Reiter, 2001a, Reiter, 2001b] to provide an argument in favor of the correctness of our approach with respect to the K-fluent based specification.

3.2 The language and basic action theories

In this section, we consider a version of the situation calculus characterized in Section 2.1.3 with the following two simplifying assumptions: 1) the language has no functional fluents, but non-fluent function symbols are permitted; 2) the language has no non-fluent predicate symbols except for the equality predicate, the predicate symbol $\sqsubset$ (it occurs in the foundational axioms to define the order between situations), and the predicate symbol $Poss$ (it occurs in precondition axioms to characterize actions possible in certain situations). Neither assumption leads to any loss of expressiveness. First, instead of a functional fluent, the axiomatizer can use a corresponding relational fluent (but should enforce via an appropriate axiomatization that the

[2][Reiter, 2001a] formulates a sufficient condition when regression is computationally efficient: it is sufficient to consider basic action theories such that successor state axioms are *context free*, i.e., right hand sides of axioms do not mention other fluents.

object argument of the fluent must always exist and be unique). Second, a non-fluent predicate symbol can be replaced by a relational fluent (the axiomatizer should add an obvious successor state axiom saying that the truth value of such a fluent persists from situation to situation no matter what action has been performed). We introduce these restrictions on the language to simplify formalization. [3]

In the sequel, we will also need the definition of an *objective situation-suppressed sentence* and the definition of a *subjective sentence*; both definitions are introduced in [Reiter, 2001b]. An objective situation-suppressed sentence is an expression without any free variables composed from situation suppressed relational fluent atoms and expressions $term_1 = term_2$ by boolean connectives and quantifiers; in a nutshell, objective situation-suppressed sentences are expressions that do not mention K and have no situational argument. For the sake of brevity, these sentences are called simply objective. A *subjective sentence about* a ground situation term σ is a sentence of the form **Knows**(β, σ), where β is objective, or it has the form $\neg W$, where W is a subjective sentence about σ, or it has the form $W_1 \vee W_2$, where W_1 and W_2 are subjective sentences about σ. In a nutshell, subjective sentences are boolean combinations of statements about what the agent knows or does not know about the objective world.

In this chapter, we will use boolean constant symbols YES and NO; informally, these symbols will correspond to values of objective sentences measured by sensors.

3.2.1 Sensing Actions

As described in [Levesque, 1996, Golden and Weld, 1996], sensing actions are those actions which the agent or robot takes to obtain information about certain properties of the external world, rather than to change them. We find convenient to associate each sense action with a value returned by a sensor at a time when the sensor gathers information from the real world: this value can be a boolean constant or a real number. More specifically, we introduce a ternary term $sense(q, v, t)$ to represent sense actions. The first argument can be either a term, e.g., a physical quantity that needs to be measured (in this case, v can be a real or rational number that represents a value of the physical quantity returned by a sensor), or an objective situation-suppressed sentence whose truth value the sensor has to detect (in this case, v can be a boolean constant representing a truth value returned by a sensor). The last argument t is a moment of

[3]The first assumption helped to simplify treatment of knowledge-based programs with sensing in [Reiter, 2001a, Reiter, 2001b]: because $read$ knowledge-producing actions sense the values of functional fluents they can be ignored if there are no functional fluents. The second assumption was introduced there for purely technical reasons.

time. We do not include axioms of real or rational numbers in any of basic theories of actions $\mathcal{D}$, but assume instead that the second and the third arguments of $sense(q, v, t)$ have a standard interpretation.

In the remainder of this chapter, we concentrate on sense actions that estimate a truth value of an objective sentence ϕ. In addition, without loss of generality, in this chapter we are going to ignore the temporal argument of sense actions because in this chapter we are more interested in issues that are easier to introduce without having to worry about explicit temporal arguments. Later, in Chapter 6 we consider sense actions with three arguments.

Informally, ordinary fluents are usually understood as properties of the world external to the mind of the agent that reasons about the world and does physical actions in the world. Consequently, the common understanding of successor state axioms formulated in Section 2.1.1 is that they characterize causal effects of physical actions on properties of the real world and that the right hand side of each axiom provides complete explanation of any change of a property in the environment in terms of physical actions executed in the environment. As for sense actions, we have seen in Section 2.1.3 that it is convenient to formulate no side-effects assumption (2.11) to obtain the successor state axiom for the epistemic fluent $K(\sigma, s)$.

Nevertheless, we propose to consider a different perspective according to which *fluents are understood as internal ('mental') representations of the properties of the real world* and successor state axioms characterize the dynamics of this internal representation in the mind of the agent. According to this new perspective, whenever the agent does a physical action in the real world or uses sensors to gather information about the external world, both physical and sense actions may have effects on the agent's internal representation of the real world, in other words, both types of actions may have effects on fluents. As a consequence, a change in a value of a fluent can be explained not only by physical actions performed by the agent, but also by data gathered by a sensor in the real world. In addition, in the general case, we cannot assume that the results of sensing are conditioned by the current 'mental model' of the agent: external forces and agents may produce changes in the real world, but those exogenous actions and events are not necessarily directly observable by the agent acting in the world. The most important example of exogenous actions that are not directly observable by the agent are actions that nature does when the agent initiates a stochastic action. However, sensing may provide the agent with information about effects of exogenous actions and events and these effects will serve as indirect indications of external events that occurred in the environment or exogenous actions that have been executed by other agents or by nature. The agent may be able to compare data measured by sensors with currently expected values of properties and, in the case when there are discrepancies, the agent may be able to find what external factors can explain the observed discrepancies. Consequently, the dynamic nature of real world environments and inability of the agent to observe directly all occurrences of exogenous actions is one of the important reasons why information gathered by sensors of the agent should be grounded in the real world and not in the current 'mental model' of the agent.[4] In brief, the agent should be able to sense the real information in the external world, not just information that the agent may expect to sense. Formally, this means that *the precondition axioms for sense actions in the general case should not constrain results of sensing by formulas that hold in the current situation*. There are cases when sensors can malfunction, or when the agent can be blinded by mis-perceptions (e.g., a wall that is perceived white in the day light will have the green color in the green light). Below, we assume that sensors function correctly and we do not consider the problem of mis-perceptions (this problem is extensively studied in [Pirri and Finzi, 1999]).

Thus, to accommodate our new perspective in terms of basic theories of actions, we can keep the syntactic form of precondition and successor state axioms intact, but we have to modify the right hand sides of successor state axioms to account for those logical conditions on sense actions that 'cause' certain effects on fluents. Together with formulas $\gamma_F^+(\vec{x}, a, s)$ and $\gamma_F^-(\vec{x}, a, s)$ that mention physical actions only (see the successor state axiom 2.6), all those new logical conditions that mention sense actions will characterize completely the dynamics of the agent's internal representation of the property F. This modification of successor state axioms will be the only change in the axiomatization that we make to be able to reason about the effects of both sense and physical actions: no other predicates have to be introduced, no other axioms have to be included in the domain theory $\mathcal{D}$ (besides precondition axioms for sense actions). We defer to the last section of this chapter any further informal discussions of the proposed perspective and comparison with other alternative approaches.[5] To summarize the proposed approach, let us consider a generic successor state axiom (2.6):

$$F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \lor F(\vec{x}, s) \land \neg\gamma_F^-(\vec{x}, a, s),$$

that mentions physical actions only, and let us assume that we have a fixed set of sense actions

[4]Another reason can be logical incompleteness of the theory that characterizes the environment: sensing can deliver new information to the agent.

[5]Note that the axiomatizer can try to account for differences between the agent's internal representation of the properties of the real world, 'objective' representation of the external world, and the agent's representation of other agents internal representations by introducing a new argument of the sort *agent* to each fluent; we do not explore this approach in this dissertation.

$sense(\phi_1, v_1)$, ..., $sense(\phi_n, v_n)$, where the first argument is like a constant that names a sensing action, but we write this name using $\phi_1, ..., \phi_n$: situation suppressed expressions that do not mention the K fluent (they can be objective sentences, but they can also have occurrences of free object variables, in the latter case, we have a sensing action term with variables as arguments). In other words, no reification is needed in these sensing actions. In addition, let the actions $sense(\phi_{i_1}, v_{i_1})$, ... , $sense(\phi_{i_k}, v_{i_k})$ make the fluent F true if in situation s conditions (they are formulas uniform in s) $\beta_1^+(\vec{x}, v_{i_1}, s)$, ... , $\beta_k^+(\vec{x}, v_{i_k}, s)$ hold, respectively, and the actions from the following subset $sense(\phi_{j_1}, v_{j_1})$, ... , $sense(\phi_{j_m}, v_{j_m})$ make the fluent F false if, respectively, conditions (they are also formulas uniform in s) $\beta_1^-(\vec{x}, v_{j_1}, s)$, ... , $\beta_m^-(\vec{x}, v_{j_m}, s)$ hold in situation s. To account for effects of sense actions we suggest to modify the successor state axiom for F:

$$\begin{aligned}F(\vec{x}, do(a,s)) \equiv\ & \gamma_F^+(\vec{x}, a, s) \lor \\ & (\exists v_{i_1}) a = \textit{sense}(\phi_{i_1}, v_{i_1}) \land \beta_1^+(\vec{x}, v_{i_1}, s) \lor \ldots \lor (\exists v_{i_k}) a = \textit{sense}(\phi_{i_k}, v_{i_k}) \land \beta_k^+(\vec{x}, v_{i_k}, s) \lor \\ & F(\vec{x}, s) \land \neg\gamma_F^-(\vec{x}, a, s) \land \\ & (\neg(\exists v_{j_1}) a = \textit{sense}(\phi_{j_1}, v_{j_1}) \land \beta_1^-(\vec{x}, v_{j_1}, s) \land \ldots \land \neg(\exists v_{j_m}) a = \textit{sense}(\phi_{j_m}, v_{j_m}) \land \beta_m^-(\vec{x}, v_{j_m}, s)),\end{aligned} \tag{3.1}$$

where we assume that sense actions mentioned in the axiom are all the sense actions that may have an effect on fluent F (this is the causal completeness assumption that is needed to solve the frame problem; a similar assumption was formulated for physical actions in [Reiter, 1991], see also Section 2.1.1 for details). The following example illustrates how modified successor state axioms can be written using sense actions.

Example 3.2.1: Consider the following successor state axiom for the fluent $broken(x)$, where x is an object that can be smashed or repaired by an agent; this axiom has occurrences of physical actions only:

$$broken(x, do(a,s)) \equiv\ a = smash(x) \lor broken(x,s) \land a \neq repair(x).$$

Let us assume that our agent has a sensor that can report whether an object is occluded or broken:

$$sense((\exists y).occluded(x,y) \lor broken(x), v_1),$$

where v_1 may have values YES or NO and $occluded(x, y, s)$ is a fluent that holds in s if x is occluded by an object y (if the agent looks at x from the current viewpoint).[6] The agent has also

[6]Note that the first argument $(\exists y).occluded(x,y) \lor broken(x)$ of the sense action is not a reified formula, but rather a name of this particular action that has two variables, x and v_1, as its arguments

an internal power source provided by an electric battery. There is no model of how the battery voltage (characterized by the fluent $voltage(v_2, s)$, where v_2 is a nominal value of the voltage) changes when the agent executes physical actions in the world, but the agent can measure the voltage any time by doing the sense action $sense(Battery, v_2)$ (we can assume that when the battery is connected to a charger, the upper limit on voltage is 24). Let $lookingAt(x, s)$ be a fluent that holds if the agent looks at the object x; then the precondition axioms for the sense actions can be

$$(\forall v_1, s).\, Poss(sense((\exists y).occluded(x, y) \lor broken(x), v_1), s) \;\equiv\; lookingAt(x, s)$$
$$(\forall v_2, s).\, Poss(sense(Battery, v_2), s) \;\equiv 0 \leq v_2 \leq 24.$$

Note that the values v_1 that can be measured by sensors are *not* constrained on the right hand side. (We omit precondition axioms for physical actions $smash(x), repair(x)$ because they are not essential for this example.)

The modified successor state axiom for $broken$ accounts for sense actions in addition to physical actions:

$$\begin{aligned}
broken(x, do(a, s)) \equiv\; & a = smash(x) \lor \\
& a = sense((\exists y).occluded(x, y) \lor broken(x), \text{YES}) \land \neg(\exists y) occluded(x, y, s) \lor \\
& broken(x, s) \land [a \neq repair(x) \land a \neq sense((\exists y).occluded(x, y) \lor broken(x), \text{NO})\,]. \\
voltage(v, do(a, s)) \equiv\; & a = sense(Battery, v) \lor voltage(v, s) \land \neg(\exists v') a = sense(Battery, v').
\end{aligned}$$

As we see from the example, if an axiomatizer works with actions that sense specific objective properties (e.g., $(\exists y).occluded(x, y) \lor broken(x)$), then the modified successor state axiom can be easily obtained from the given successor state axiom for a fluent, but in the case of sensing actions $sense(\phi, v)$ that have to detect a truth value v of an unspecified ϕ, we can describe only a generic form of a modified successor state axiom: the precise form of expressions $\beta_1^-(\vec{x}, v_{j_1}, s), \ldots, \beta_m^-(\vec{x}, v_{j_m}, s)$ and $\beta_1^+(\vec{x}, v_{i_1}, s), \ldots, \beta_k^+(\vec{x}, v_{i_k}, s)$ is domain dependent. However, for one particular case, when ϕ is a situation suppressed fluent expression, it is possible to specify precisely all modified successor state axioms. This case is also interesting because for sense actions of this restricted form, we can establish the correspondence between solutions of the forward projection problem formulated in [Reiter, 2001b] and in [Soutchanski, 2001a].

For this reason, we introduce here two mutually exclusive sets of sense actions. One of them will be used in theories with the K fluent, another set of sensing actions with a run time argument will be used in theories without the K fluent. In the first set (see [Reiter, 2001b]), all sense actions have the form $sense_\psi(\vec{x})$, where $\psi(\vec{x})$ is an objective situation-suppressed expression. Below we pay attention to a strict subset of this set: sense-fluent actions which are a special case of previously mentioned sensing actions. Thus, we consider only sense actions where $\psi(\vec{x})$ is a situation-suppressed relational fluent symbol: $sense_{F_1}(\vec{x}_1)$, ..., $sense_{F_m}(\vec{x}_m)$, where m is the number of fluents (we call them *usual sensing actions*). The second set consists of all those substitutions of a binary action term $sense(q, v)$, which have a situation-suppressed relational fluent atom as their first argument and a truth value constant as their second argument (we call these *run-time sensing actions*).[7] So, we consider two distinct subsets of run-time sensing actions and we consider m distinct action terms in each subset: $sense(F_1(\vec{x}_1), v)$,...,$sense(F_m(\vec{x}_m), v)$, where $v \in \{\text{YES}, \text{NO}\}$.[8]

In the next subsection, we formulate the syntactic form of two types of basic action theories (each of them uses its own set of sense actions and either has or does not have an epistemic fluent $K(\sigma, s)$), then we formulate the forward projection task in each of these basic action theories, and finally, we formulate and prove the correspondence between solutions of these two different forward projection tasks.

3.2.2 Basic Action Theories

We consider two types of basic action theories:

$$\mathcal{D}^K = \mathcal{D}_{ss}^K \cup \mathcal{D}_{ap}^K \cup \mathcal{D}_{S_0}^K \cup \mathcal{K}_{Init} \cup \Sigma^K \cup \mathcal{D}_{una}$$
$$\mathcal{D} = \mathcal{D}_{ss} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{S_0} \cup \Sigma \cup \mathcal{D}_{una}$$

The differences and correspondences between $\mathcal{D}^K$ and $\mathcal{D}$ are the following. Sense actions $sense(F_i(\vec{x}_i), v)$ may occur only in $\mathcal{D}$ and sense actions $sense_{F_i}(\vec{x}_i)$ may occur only in $\mathcal{D}^K$; a physical action $A(\vec{x})$ occurs in $\mathcal{D}$ if and only if it occurs in $\mathcal{D}^K$.

The theory $\mathcal{D}^K$ above is a basic action theory with knowledge as characterized in Section 2.1.3. In particular, a set $\mathcal{D}_{ss}^K$ includes successor state axioms that have the syntactic form (2.6) and mention physical actions only; this set includes also the successor state axiom (2.14) for the epistemic fluent $K(s', s)$ that have only occurrences of sense actions. In contrast, the set $\mathcal{D}_{ss}$ does **not** include a successor state axiom for K fluent. As we discussed in Section 2.1.1, in the general case the successor state axioms are logically equivalent to the conjunction of both effect axioms and state constraints (2.7), if the causal completeness assumption holds. In this general case, the modified successor state axioms in $\mathcal{D}_{ss}$ can be quite intricate (we

[7] Notice that we do not allow more expressive sense actions which may have quantifiers over object variables in objective expression q.

[8] Thus, we assume that all fluents are directly observable. We would have less sense actions if some fluents were not directly observable, i.e., data about their truth values were not available to sensors.

consider an example in the next section). Let us consider here only the simpler case when the domain theory has no state constraints, i.e., there is no uniform situation calculus formula $\phi(s)$ with s as its only free variable such that the universal closure $\forall s.\phi(s)$ is entailed by $\mathcal{D}_{ss}$ (because logically valid sentences are trivially entailed by $\mathcal{D}_{ss}$ we assume here that $\phi(s)$ is not a logically valid sentence). In this case, there are no causal dependencies between fluents[9] , all fluents evolve independently from each other and the axiom (3.1) has the much simpler form (3.2) below. All successor state axioms in $\mathcal{D}_{ss}$ mention both physical and sense actions (recall that only sense actions $sense(F_i(\vec{x}_i), v)$ may occur in $\mathcal{D}_{ss}$). Every axiom is obtained by the following transformation from the corresponding successor state axiom (2.6) in $\mathcal{D}_{ss}^K$:

$$\begin{aligned} F(\vec{x}, do(a,s)) \equiv & \\ & \gamma_F^+(\vec{x}, a, s) \ \lor \ a = sense(F(\vec{x}), \text{YES}) \ \lor \\ & F(\vec{x}, s) \land (\, \neg\gamma_F^-(\vec{x}, a, s) \land a \neq sense(F(\vec{x}), \text{NO})\,). \end{aligned} \tag{3.2}$$

In other words, the fluent $F(\vec{x}, do(a,s))$ is true iff a is a physical action and the condition $\gamma_F^+(\vec{x}, a, s)$ holds or a is a sense action and the sensors tell that the fluent is true or the fluent was true in s and neither $\gamma_F^-(\vec{x}, a, s)$ is true nor a is a sense action determining that the fluent is false.

$\mathcal{D}_{ap}$ is a set of action precondition axioms, one for each primitive action $A(\vec{x})$, having the syntactic form

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s),$$

where $\Pi_A(\vec{x}, s)$ is an objective formula with free variables among $\vec{x}, s$, and whose only situation term is s. Action precondition axioms characterize (via the formula $\Pi_A(\vec{x}, s)$) the conditions under which it is possible to execute action $A(\vec{x})$ in situation s. If $A(\vec{x})$ is an action $sense(F_i(\vec{x_i}), v)$, then the right hand side of this axiom does not mention v: as we have argued in the previous section, any boolean value v must be possible in any situation s when the agent executes a sense action $sense(F_i(\vec{x}_i), v)$.

$\mathcal{D}_{ap}^K$ is also a set of action precondition axioms. These coincide with axioms in $\mathcal{D}_{ap}$ for physical and sense actions, i.e., they may not mention subjective expressions with **Knows**(ϕ, s).

$\mathcal{D}_{S_0}^K$ is a set of first order sentences whose only situation terms are S_0, and they are all of the form **Knows**(κ_i, S_0), $i = 1, \ldots, n$, where each κ_i is objective. In other words, the initial database consists exclusively of sentences declaring what the agent knows about the world

[9]This means also that under the causal completeness assumption, the successor state axioms are logically equivalent to positive and negative effect axioms only.

she inhabits, but there are no objective sentences, and there are no sentences declaring what the agent knows about what she knows. Note that beside the restriction that each κ_i is objective, formulas κ_i can be arbitrary and may include disjunctions, existential quantifiers and any boolean combinations of fluents. This means that the agent may have incomplete knowledge about the world, but the theory completely characterizes what incomplete knowledge the agent has (e.g., **Knows**$(P, S_0) \vee$ **Knows**(Q, S_0) is not allowed). Note that there can be fluents about which nothing is known in the initial situation. So we can simply suppose that $\mathcal{D}^K_{S_0}$ consists of a single sentence of the form **Knows**(κ, S_0), where κ is objective (possibly, it is a conjunction of other objective sentences). Then, $\mathcal{D}_{S_0} = \kappa_1[S_0] \wedge \cdots \wedge \kappa_n[S_0]$, i.e., we can simply suppose that a special class of initial databases consists of a single objective sentence of the form $\kappa[S_0]$. Note how $\mathcal{D}_{S_0}$ and $\mathcal{D}^K_{S_0}$ correspond to each other; we consider this special class of initial databases because it is considered in [Reiter, 2001b] and we are interested in establishing the correspondence between [Reiter, 2001b] and [Soutchanski, 2001a].[10]

$\mathcal{K}_{init}$ includes the reflexivity axiom (2.18); as we will see, in this case Lemma 3.4.9 holds. Theories like $\mathcal{D}^K$ characterized by all these assumptions are called *epistemically accurate theories* in [De Giacomo *et al.*, 2002, Sardina *et al.*, submitted], meaning that what is known by the agent accurately reflects what the theory says about the dynamic system.

$\mathcal{D}_{una}$ is the set of unique names axioms for actions. These will include unique names axioms for sense actions.

3.3 A blocks world example

In this section, we consider an example of reasoning about effects of sequences of actions using theories $\mathcal{D}$ and $\mathcal{D}^{\mathcal{K}}$. In particular, we rely on a standard axiomatization of the blocks world with successor state and action precondition axioms considered in 2.1.1, but we add an agent argument to all physical action terms. Assume that there is a robot who wants to put all blocks on the table (initially, there may be some towers of blocks) and possibly there are several other agents (who act independently and can also move blocks from one location to another). The robot is not able to observe occurrences of their actions directly, but the robot can notice effects of their actions by sensing values of fluents. We leave sense actions without an agent argument because we attribute them to the robot only. We use the following function

[10] Another reason for considering $\mathcal{D}^K_{S_0}$ of this special form is that it seems reasonable to start with an epistemically uniform initial theory and pay attention to epistemically correct theories by introducing the axiom of reflexivity, see Lemma 3.4.9.

and predicate constants.

Fluents

- $On(x, y, s)$: Block x is on block y, in situation s.
- $Clear(x, s)$: Block x has no other blocks on top of it, in situation s.
- $Ontable(x, s)$: Block x is on the table in s.

Actions

- $move(g, x, y)$: Agent g moves block x onto block y, provided both are clear.
- $moveToTable(g, x)$: Agent g moves block x onto the table, provided x is clear and is not on the table.
- $sense_{Ontable}(x)$: Sense whether block x is on the table.
- $sense_{Clear}(x)$: Sense whether block x is clear.
- $sense_{On}(x, y)$: Sense whether block x is on top of y.
  These last three 'usual' sensing actions can occur only in $\mathcal{D}^K$. The outcome of all aforementioned sense actions can be either "yes" or "no". In the case when the outcome of a sense action is "yes", we would like to add knowledge about the fluent corresponding to the subscript of the sense action to the original theory $\mathcal{D}^K$. Object arguments of this fluent coincide with object arguments of the sense action, and the situational argument is determined by the situation when the sense action has been executed. In the case when the outcome of a sense action is "no", we would like to add knowledge about the negation of the fluent corresponding to the subscript of the sense action to the original theory $\mathcal{D}^K$ (arguments of the fluent are determined similarly to the previous case). By supplementing $\mathcal{D}^K$ with new formulas, we are able to repeatedly use information obtained from sensors in subsequent reasoning. The following three 'run-time' sensing actions can occur only in $\mathcal{D}$.
- $sense(Ontable(x), v)$: Connect to a sensor, if block x is on the table, then v has the value YES, if not, then v is NO.

- $sense(Clear(x), v)$: Ask a sensor whether the block x is clear; if it is, then the value v is YES, otherwise v is NO.

- $sense(On(x, y), v)$: Query a sensor whether x is on top of y; if it is, then v is YES, otherwise v is NO.

We start our examples with considering the theory $\mathcal{D}^{\mathcal{K}}$ and later in this section we consider the theory $\mathcal{D}$. The theory $\mathcal{D}^{\mathcal{K}}$ includes the standard set of unique name axioms for actions and the following axioms.

Successor state axioms (D_{ss}^K)

$$On(x, y, do(a, s)) \equiv (\exists g).\, a{=}move(g, x, y) \vee \\ On(x, y, s) \wedge \neg(\exists g).\, a{=}moveToTable(g, x) \wedge \neg(\exists g, z)\, a{=}move(g, x, z).$$

$$Ontable(x, do(a, s)) \equiv (\exists g).a{=}moveToTable(g, x) \vee \\ Ontable(x, s) \wedge \neg(\exists g, y).\, a{=}move(g, x, y).$$

$$Clear(x, do(a, s)) \equiv (\exists g, y, z).[\, a{=}move(g, y, z) \vee a{=}moveToTable(g, y)\,] \wedge On(y, x, s) \vee \\ Clear(x, s) \wedge \neg(\exists g, w)\, a = move(g, w, x).$$

Note that these axioms have no occurrences of sense actions. A version of the axiom (2.17) is also included in D_{ss}^K: it mentions only actions $sense_{Ontable}(x)$, $sense_{Clear}(x)$, $sense_{On}(x, y)$.

Action precondition axioms (D_{ap}^K)

$$Poss(move(g, x, y), s) \equiv Clear(x, s) \wedge Clear(y, s) \wedge x \neq y.$$

$$Poss(moveToTable(g, x), s) \equiv Clear(x, s) \wedge \neg Ontable(x, s).$$

In this example, all sense actions are always possible, i.e., the right hand sides of precondition axioms for sense actions are *true*.

Let the initial situation be such that **Knows**$(Clear(B), S_0)$, nothing else is known about which blocks are where and whether they are clear or not. In addition, there are state constraints associated with the blocks world, and we must suppose the robot knows that these constraints hold initially[11]:

Knows$((\forall x, y).On(x, y) \supset \neg On(y, x), S_0)$,

[11] As we mentioned in Example 2.1.1, one can prove by induction over situations that these state constraints hold in any situation s.

Knows$((\forall x, y, z).On(y, x) \wedge On(z, x) \supset y = z, S_0)$,
Knows$((\forall x, y, z).On(x, y) \wedge On(x, z) \supset y = z, S_0)$,
Knows$((\forall x).Ontable(x) \equiv \neg(\exists y)On(x, y), S_0)$,
Knows$((\forall x).Clear(x) \equiv \neg(\exists y)On(y, x), S_0)$.

The conjunction of axioms about the initial situation is **Knows**(κ, S_0). Because this set of axioms is weak (it does not even entail whether block B is on the table or not), we are intending to augment $\mathcal{D}^{\mathcal{K}}$ with axioms representing sensory information whenever the robot does one of sense actions.

To illustrate reasoning about effects of both physical and sense actions we consider two different scenarios. The first scenario provides an example of 'belief expansion' (beliefs change as the result of new information acquired). The second scenario provides an example of 'belief update' (beliefs change as the result of the realization that the world has changed).

First scenario.

In the theory $\mathcal{D}^K$ we are primarily interested in solving the forward projection tasks of the form $\mathcal{D}^K \models$ **Knows**(ψ, S), where ψ is objective and S is a ground situation term. Because $\mathcal{D}^K_{S_0}$ includes neither **Knows**$(\neg Clear(A), S_0)$, nor **Knows**$(On(B, A), S_0)$, nor **Knows**$(\neg On(B, A), S_0)$, one can easily observe that $\mathcal{D}^K \not\models$ **Knows**$(\neg Clear(A), S_0)$, $\mathcal{D}^K \not\models$ **Knows**$(On(B, A), S_0)$, $\mathcal{D}^K \not\models$ **Knows**$(\neg On(B, A), S_0)$. Suppose that the robot R does the sensing action $sense_{On}(B, A)$ to determine whether the block B is on top of the block A, and suppose that the sensor replies 'yes', then we add **Knows**$(On(B, A), do(sense_{On}(B, A), S_0))$ to $\mathcal{D}^{\mathcal{K}}$. This is a case of belief expansion. This properly augmented set of axioms is now sufficient to solve the following set of the forward projection tasks:

$\mathcal{D}^K \cup$ **Knows**$(On(B, A), do(sense_{On}(B, A), S_0)) \models$
Knows$(\neg Clear(A), do(sense_{On}(B, A), S_0))$,

$\mathcal{D}^K \cup$ **Knows**$(On(B, A), do(sense_{On}(B, A), S_0)) \models$
Knows$(Clear(A), do(moveToTable(R, B), do(sense_{On}(B, A), S_0)))$,

$\mathcal{D}^K \cup$ **Knows**$(On(B, A), do(sense_{On}(B, A), S_0)) \models$
Knows$(\neg On(B, A), do(moveToTable(R, B), do(sense_{On}(B, A), S_0)))$.

Second scenario.

Let the robot do the following sequence of actions. First, the robot senses whether B is on the table or not; suppose, that B is not on the table. In this case, because it is not known initially where B is located we want to add the axiom **Knows**$(\neg Ontable(B), do(sense_{Ontable}(B), S_0))$

to $\mathcal{D}^K$: this axiom will supplement axioms about the initial situation. In the sequel, we denote $S_N = do(sense_{Ontable}(B), S_0))$. Second, the robot R moves the block B to the table; let's denote the resulting situation $S_M = do(moveToTable(R, B), S_N)$. We are interested in solving the following entailment task:

$$\mathcal{D}^{\mathcal{K}} \wedge \mathbf{Knows}(\neg Ontable(B), S_N) \models \mathbf{Knows}(Ontable(B), S_M). \tag{3.3}$$

One can prove that the formula at the right hand side is entailed by axioms on the left hand side. Now, suppose that someone else moves block B from the table onto another block and suppose that the robot did not observe the occurrence of this action (it was looking elsewhere). In other words, from the robot's perspective the current situation is still S_M. This means that in the second scenario we allow unobservable exogenous actions and the robot works in a partially observable environment. At this moment, if the robot does the sense action $sense_{Ontable}(B)$, then the outcome from sensors is 'no'. One can prove

$$\mathbf{Knows}(\neg Ontable(B), do(sense_{Ontable}(B), S_M))$$

is inconsistent with $\mathbf{Knows}(Ontable(B), do(sense_{Ontable}(B), S_M))$ which is a logical consequence of (3.3) and the successor state axiom for knowledge (2.14).

Let us consider now the theory $\mathcal{D}$. It includes the following set of successor state axioms.

$$\begin{array}{l}
On(x, y, do(a, s)) \equiv \exists g\,(a = move(g, x, y)) \vee a = sense(On(x, y), \text{YES}) \vee \\
\quad On(x, y, s) \wedge \neg\exists g\,(a = moveToTable(g, x)) \wedge \neg\exists g, z\,(a = move(g, x, z)) \wedge \\
\qquad a \neq sense(On(x, y), \text{NO}) \wedge \\
\qquad\quad \neg(\exists z).[z \neq x \wedge a = sense(On(z, y), \text{YES})] \wedge \neg(\exists w).[w \neq y \wedge a = sense(On(x, w), \text{YES})] \wedge \\
\qquad\qquad a \neq sense(Ontable(x), \text{YES}) \wedge a \neq sense(Clear(y), \text{YES}).
\end{array}$$

$$\begin{array}{l}
Ontable(x, do(a, s)) \equiv \exists g\,(a = moveToTable(g, x)) \vee \\
\qquad\qquad a = sense(Ontable(x), \text{YES}) \vee \\
\qquad Ontable(x, s) \wedge \neg(\exists g, y).\, a = move(g, x, y) \wedge \\
\qquad\quad a \neq sense(Ontable(x), \text{NO}) \wedge \neg(\exists w).a = sense(On(x, w), \text{YES}).
\end{array}$$

$$\begin{array}{l}
Clear(x, do(a, s)) \equiv (\exists g, y, z).[\,a = move(g, y, z) \vee a = moveToTable(g, y)\,] \wedge On(y, x, s) \vee \\
\qquad a = sense(Clear(x), \text{YES}) \vee \\
\quad Clear(x, s) \wedge \neg\exists g, w\,(a = move(g, w, x)) \wedge \\
\qquad a \neq sense(Clear(x), \text{NO}) \wedge \neg(\exists y).a = sense(On(y, x), \text{YES}).
\end{array}$$

The set $\mathcal{D}_{ss}$ does not include the successor state axiom (2.17) for the K fluent. The set of precondition axioms $\mathcal{D}_{ap}$ coincides with $\mathcal{D}^K_{ap}$; D also includes the set of unique names axioms for actions. Let $\mathcal{D}_{S_0}$ includes $C(S_0)$, these are state constraints mentioned in the example (2.1.1), and $Clear(B, S_0)$, nothing else is known about the initial locations of blocks. In $\mathcal{D}$, we are interested in the solution of the forward projection task of the form $\mathcal{D} \models \psi(S)$, where S is a ground situation term. We would like to compare the solution of this task in $\mathcal{D}$ with the solution in $\mathcal{D}^K$, for this reason we consider the same two scenarios.

First scenario.

Obviously, $\mathcal{D} \not\models Clear(A, S_0)$, $\mathcal{D} \not\models On(B, A, S_0)$, $\mathcal{D} \not\models \neg On(B, A, S_0)$. Suppose that the robot R senses whether B is on A and finds that this is true. One can prove the following entailments:

$$\begin{aligned}
\mathcal{D} &\models \neg Clear(A, do(sense(On(B, A), \text{YES}), S_0)), \\
\mathcal{D} &\models \; Clear(A, do(moveToTable(R, B), do(sense(On(B, A), \text{YES}), S_0))), \\
\mathcal{D} &\models \; \neg On(B, A, do(moveToTable(R, B), do(sense(On(B, A), \text{YES}), S_0))).
\end{aligned}$$

Notice that we can solve all these forward projection tasks without augmenting the given set of axioms $\mathcal{D}$.

Second scenario.

Consider again the same sequence of actions and again suppose that initially B is not on the table. First, the robot does $sense(Ontable(B), \text{NO})$, in other words, it determines that B is not on the table; we denote $do(sense(Ontable(B), \text{NO}), S_0)$ by S'_n. Second, the robot moves B to the table. Clearly,

$$\mathcal{D} \models Ontable(B, do(moveToTable(R, B), S'_n)).$$

In the sequel, we denote $do(moveToTable(R, B), S'_n)$ by S'_m. Similarly to the previous scenario, let someone move B from the table, assume that this action is not observable, i.e., the robot thinks that the current situation is still S'_m. Now the robot does $sense(Ontable(B), \text{NO})$ in the situation S'_m; one can prove that

$$\mathcal{D} \models \neg Ontable(B, do(sense(Ontable(B), \text{NO}), S'_m)).$$

Thus, we can obtain correct entailments from $\mathcal{D}$ even if occurrences of exogenous actions are not observable.[12] We see that 'beliefs' of the robot have been updated, unlike in $\mathcal{D}^K$.

[12] The last entailment can be considered as an indication that the actual situation just before doing the second sense action is different from S'_m. But the task of determining what physical actions occurred between S'_m and the moment when the robot did the second sense action remains outside the scope of the approach presented in this chapter. A solution to this task is necessary to compute the corrected situation term that includes occurrences of exogenous physical actions. The approach to solving diagnostic problems suggested in [McIlraith, 1998] can be used to compute actual situation terms.

In the subsequent sections, we consider only the case when axioms corresponding to new sensory information are consistent with previously obtained set of axioms: we make this simplifying assumption because we are primarily interested in establishing the correspondence between reasoning about effects of actions in $\mathcal{D}^{\mathcal{K}}$ and reasoning about effects of actions in $\mathcal{D}$.

3.4 The forward projection task

In this section, our main goal is to find the connection between solutions to the forward projection task in the two different frameworks outlined above. First, we consider this task when $\mathcal{D}$ is the domain axiomatization and sense actions have the form $sense(F_i(\vec{x}), v_i)$. Second, we consider this task when the domain axiomatization is $\mathcal{D}^K$ and sense actions have the form $sense_{F_i}(\vec{x})$.

As we have seen in Section 2.2.1, when $\mathcal{T}$ is a domain theory without knowledge and sensing, after repeatedly applying the regression operator $\mathcal{R}$ a necessary number of times, the original projection task reduces to first order theorem proving in the initial database $\mathcal{T}_{S_0}$ together with unique names axioms for actions T_{una}:

Theorem 3.4.1: (The Regression Theorem, [Reiter, 2001a, Pirri and Reiter, 1999]) *Suppose that $\mathcal{T}$ is a basic action theory (without knowledge), and that $\phi(S)$ is a regressable sentence, where S is a ground situation term that mentions physical actions only. Then*

$\mathcal{T} \models \phi(S)$ *iff* $\mathcal{T}_{una} \cup \mathcal{T}_{S_0} \models \mathcal{R}_{\mathcal{T}}[\phi(S)]$

Note that $\mathcal{R}_{\mathcal{T}}[\phi(S)]$ can mention only situation independent atoms and fluent atoms whose only situation argument is S_0. Observe also that the regression operator is defined relative to the set of successor state and precondition axioms in $\mathcal{T}$: if one varies these axioms, the result of regression varies as well.

3.4.1 The theory $\mathcal{D}$

Let a basic action theory $\mathcal{T}$ with sense actions have the form $\mathcal{D}$. It is surprisingly straightforward to use regression in this setting to solve the forward projection task if ϕ is an objective regressable sentence and we are given a ground situation term S_1 that may mention both physical and sensing actions (recall that the second argument of all sense actions mentioned in $\mathcal{D}$ is either the constant YES or the constant NO). No modifications of the regression operator are required and the regression theorem 3.4.1 applies also in this setting:

Theorem 3.4.2: (The Regression Theorem in $\mathcal{D}$)
Suppose that $\mathcal{D}$ is a basic action theory, and that $\phi(S_1)$ is a regressable sentence that does not mention K. Then
$\mathcal{D} \models \phi(S_1)$ *iff* $\mathcal{D}_{una} \cup \mathcal{D}_{S_0} \models \mathcal{R}_{\mathcal{D}}[\phi(S_1)]$

The subscript in $\mathcal{R}_{\mathcal{D}}$ indicates that the regression operator depends on $\mathcal{D}$; in particular, the regression operator eliminates all occurrences (if any) of fluents with complex situation terms $do(sense(F_i, V), S')$ using axioms (3.2). Note that the same proof as in the previous Theorem 3.4.1 applies in this case because sensing actions are the usual action terms (i.e., no reification is needed).

Note the price that we pay to enjoy this solution: no occurrences of **Knows** are permitted in the formula $\phi(S)$. Hence, if an agent needs to evaluate a formula that mixes objective and subjective sentences, the theorem 3.4.2 cannot be applied. However, as long as the agent needs to reason only about the world, but does not need to reflect about knowledge of the world that she possesses, this theorem remains useful.

3.4.2 The theory $\mathcal{D}^K$

Let a basic action theory with sense actions $\mathcal{T}$ have the form $\mathcal{D}^K$. Let ground situation term S mention both physical actions $A(\vec{Y})$ and sense actions $sense_{F_i}(\vec{X})$. Assume that all sense actions in S are useful, i.e., they provide consistent new information. Formally, this means that for every subterm $do(sense_{F_i}(\vec{X}), S') \sqsubseteq S$ of S we have:

$$\mathcal{D}^K \not\models \textbf{Knows}(F_i(\vec{X}), do(sense_{F_i}(\vec{X}), S'))$$
$$\mathcal{D}^K \not\models \textbf{Knows}(\neg F_i(\vec{X}), do(sense_{F_i}(\vec{X}), S'))$$

(every sense action may have only two outcomes: either it determines that a fluent is true, or it determines that a fluent is false). Under this reasonable assumption, we no longer can solve the projection problem for a formula $\phi(do(sense_{F_i}(\vec{X}), S))$ if ϕ includes an atom **Knows** about the fluent F_i: our axioms are not strong enough to entail whether the agent knows F_i. To solve the projection problem we need to extend our basic action theory by axioms about outcomes of sense actions. Consequently, we have to define the result of augmenting $\mathcal{D}^K$ with knowledge about the outcomes of all the sense actions occurring in S. To do this, we need a concept of the *sense outcomes* of a given action sequence S.

Definition 3.4.3: Sense outcome function [13]

[13] Both the notation and results formulated in the rest of this sub-section are introduced in [Reiter, 2001a,

A *sense outcome function* is any mapping Ω from ground situation terms to knowledge sentences, such that:

1. $\Omega(S_0) = \{\,\}$,

2. If α is not a sense action,

$$\Omega(do(\alpha, S)) = \Omega(S).$$

3. If α is a sense action $sense_\psi(\vec{X})$,

$$\Omega(do(\alpha, S)) = \Omega(S) \cup \{\mathbf{Knows}(\psi(\vec{X}), do(sense_\psi(\vec{X}), S))\} \text{ or}$$

$$\Omega(do(\alpha, S)) = \Omega(S) \cup \{\mathbf{Knows}(\neg\psi(\vec{X}), do(sense_\psi(\vec{X}), S))\}.$$

In the sequel, we shall be interested in $\mathcal{D}^K \cup \Omega(S)$, namely, the original basic action theory $\mathcal{D}^K$, augmented by knowledge about the outcomes of all sense actions in the action history S, according to the sense outcome function Ω. Because we are intending to use regression as a solution to the forward projection problem and obtain a version of the regression theorem for augmented basic action theories we need also a definition of $\Omega^\rho(S)$ - a regressed version of $\Omega(S)$.

According to Definition 2.2.8, when $\psi(\vec{x})$ is an objective expression, and σ a situation term, then $\psi(\vec{x})[\sigma]$ is that situation calculus formula resulting from adding an extra situation argument σ to every situation-suppressed fluent mentioned by ψ. According to Definition 2.2.2, $\mathcal{R}[W]$ is a situation calculus formula (whose only situation term is S_0) obtained by applying the regression operator to a regressable sentence W. Thus, $\mathcal{R}[\psi(\vec{x})[\sigma]]$ is a situation calculus formula obtained by applying the regression operator to the formula $\psi(\vec{x})[\sigma]$. Recall also that $\rho(\phi, \sigma)$ is a 'multi-step' regression operator that transforms ϕ through action terms mentioned in σ to a simpler expression without occurrences of $do(\cdot, \cdot)$, and that by Lemma 2.2.12 $\rho(\phi, \sigma)[S_0]$ and $\mathcal{R}[\,\phi[\sigma]]$ are logically equivalent relative to the given basic action theory.

Suppose Ω is a sense outcome function, σ is a ground situation and σ' is a sub-term of σ.

$$\Omega^\rho(\sigma) = \bigwedge\{\rho(\psi(\vec{X}), \sigma') \mid \mathbf{Knows}(\psi(\vec{X}), do(sense_\psi(\vec{X}), \sigma')) \in \Omega(\sigma)\} \wedge$$
$$\bigwedge\{\rho(\neg\psi(\vec{X}), \sigma') \mid \mathbf{Knows}(\neg\psi(\vec{X}), do(sense_\psi(\vec{X}), \sigma')) \in \Omega(\sigma)\},$$

where both conjunctions are over all sub-terms σ' of σ such that $\Omega(do(sense_\psi(\vec{X}), \sigma'))$ contains a subjective sentence. By convention, $\Omega^\rho(\sigma) = true$ when $\Omega(\sigma) = \{\ \}$. Notice that

Reiter, 2001b] where the reader can find additional details.

$\Omega^\rho(\sigma)$ is a situation-suppressed sentence and that 'multi-step' regression operator $\rho(\phi, \sigma)$ is understood here with respect to successor state axioms of the theory $\mathcal{D}^K$.

The following lemma is a consequence of Corollary 3.4.9, Theorem 2.2.4 and Lemma 2.2.12:

Lemma 3.4.4: *[Reiter, 2001a, Reiter, 2001b] When $\mathcal{K}_{Init}$ includes the reflexivity axiom (2.18),*

$$\mathcal{D}^K \models \Omega^\rho(\sigma)[S_0] \equiv \bigwedge_{(sense\ actions\ in\ \sigma)} \Omega(\sigma),$$

in other words, $\Omega^\rho(\sigma)[S_0]$ and the conjunction of the sentences in $\Omega(\sigma)$ are logically equivalent, relative to $\mathcal{D}^K$.

[Reiter, 2001a, Reiter, 2001b] introduce the following technical definition as a condition on a basic action theory. Suppose T is any situation calculus theory. We say that T decides all equality sentences iff for any sentence β over the language of T whose only predicate symbol is equality, $T \models \beta$ or $T \models \neg\beta$.

The following crucial theorem (see Theorem 11.7.14 in [Reiter, 2001a]) asserts that the forward projection task can be also solved in $\mathcal{D}^K$ for a formula **Knows**(ϕ, σ), when ϕ is an objective sentence. This theorem is a consequence of the regression theorem (3.4.1) and Lemma 3.4.4. Recall that $\mathcal{D}^K_{S_0} =$ **Knows**(κ, S_0), $\mathcal{D}_{S_0} = \kappa[S_0]$, where κ is objective and $\mathcal{K}_{Init}$ includes the reflexivity axiom.

Theorem 3.4.5: *[Reiter, 2001a, Reiter, 2001b] (The Regression Theorem in $\mathcal{D}^K$).*
Suppose that ϕ is an objective regressable sentence, $\mathcal{D}_{una} \cup \{\kappa[S_0]\}$ decides all equality sentences, S_2 is a ground situation term. Then

$$\mathcal{D}^K \cup \Omega(S_2) \models \textbf{Knows}(\phi, S_2) \quad \textit{iff} \qquad \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \Omega^\rho(S_2)[S_0] \models \mathcal{R}_{\mathcal{K}}[\phi[S_2]].$$

Note that in this theorem the regression operator $\mathcal{R}_{\mathcal{K}}$ is defined with respect to $\mathcal{D}^{\mathcal{K}}$. Using this theorem, one can reduce reasoning about what an agent knows about a future situation S_2 to an entailment task with respect to a much simpler set of first order axioms that may have occurrences of S_0 only; neither foundational axioms, nor precondition and successor state axioms are required to solve the entailment task. It is also remarkable that reasoning about the agent's knowledge can be accomplished mechanically (under assumptions stated above) by theorem proving that does not require manipulations with **Knows**. This has important computational advantages, because the forward projection task can be solved without reasoning about possible worlds.

3.4.3 Some Metatheoretic Properties of Basic Action Theories with Knowledge

The following propositions and lemmas formulated and proved in [Reiter, 2001a, Reiter, 2001b] will be useful in the sequel. They provide connections between knowledge about a formula in two consecutive situations or connections between knowledge about an objective formula and the formula itself (outside of epistemic context). Some of the following proofs are taken verbatim from Reiter, some other proofs are minor variations of his proofs (they are included here to keep this chapter self-contained).

Proposition 3.4.6: *Suppose $W(\vec{x})$ is a situation suppressed expression with free variables $\vec{x}$, and $A(\vec{y})$ is not a sense action. Then,*

$$\mathcal{D}^K \models (\forall \vec{x}, \vec{y}, s).\mathbf{Knows}(W(\vec{x}), do(A(\vec{y}), s)) \equiv \mathbf{Knows}(\rho^1(W(\vec{x}), A(\vec{y})), s).$$

Proof: From Lemma 2.2.10, it follows that when $W(\vec{x})$ is an objective situation suppressed expression,

$$\mathcal{D}^K \models (\forall \vec{x}, \vec{y}, s).W(\vec{x}, do(A(\vec{y}), s)) \equiv \rho^1(W(\vec{x}), A(\vec{y}))[s].$$

To complete the proof, consider the successor state axiom for K. ∎

Proposition 3.4.7: *Suppose $W(\vec{x})$ is a situation suppressed expression with free variables $\vec{x}$, and $sense_\psi(\vec{y})$ is a sense action. Then,*

$$\begin{aligned}\mathcal{D}^K \models (\forall \vec{x}, \vec{y}, s).&\mathbf{Knows}(W(\vec{x}), do(sense_\psi(\vec{y}), s)) \equiv \\ &\{\psi(\vec{y}, s) \supset \mathbf{Knows}(\psi(\vec{y}) \supset \rho^1(W(\vec{x}), sense_\psi(\vec{y})), s)\} \wedge \\ &\{\neg\psi(\vec{y}, s) \supset \mathbf{Knows}(\neg\psi(\vec{y}) \supset \rho^1(W(\vec{x}), sense_\psi(\vec{y}), s)\}.\end{aligned}$$

Proof: By the no side-effects assumption for sensing actions (2.11)

$$\mathcal{D}^K \models (\forall \vec{x}, \vec{y}, s).W(\vec{x}, do(sense_\psi(\vec{y}), s)) \equiv W(\vec{x}, s).$$

Now we can use the successor state axiom for K. ∎

Lemma 3.4.8:

$$\mathcal{D}^K \models (\forall \vec{y}, s).\mathbf{Knows}(\psi(\vec{y}), do(sense_\psi(\vec{y}), s)) \equiv \psi(\vec{y}, s) \vee \mathbf{Knows}(\psi(\vec{y}), s),$$

$$\mathcal{D}^K \models (\forall \vec{y}, s).\mathbf{Knows}(\neg\psi(\vec{y}), do(sense_\psi(\vec{y}), s)) \equiv \neg\psi(\vec{y}, s) \vee \mathbf{Knows}(\neg\psi(\vec{y}), s).$$

Proof: Take W to be ψ, then $\neg\psi$, in Proposition 3.4.7, and use the no side-effects assumption for sense actions (2.11). ∎

Corollary 3.4.9: *When $\mathcal{K}_{Init}$ includes the reflexivity axiom,*

$$\mathcal{D}^K \models (\forall \vec{y}, s).\mathbf{Knows}(\psi(\vec{y}), do(sense_\psi(\vec{y}), s)) \equiv \psi(\vec{y}, s),$$

$$\mathcal{D}^K \models (\forall \vec{y}, s).\mathbf{Knows}(\neg\psi(\vec{y}), do(sense_\psi(\vec{y}), s)) \equiv \neg\psi(\vec{y}, s).$$

Proof: We know from (2.18) that when reflexivity holds in the initial situation, it holds in all situations, so that what is known in s is true in s: $(\forall s)\mathbf{Knows}(\psi, s) \supset \psi[s]$. Next, we can use Lemma 3.4.8 and consider three cases: when $\mathbf{Knows}(\psi, s)$ is true in a model of $\mathcal{D}^K$, when $\mathbf{Knows}(\neg\psi, s)$ is true in a model of $\mathcal{D}^K$ and when both formulas are false. ∎

Lemma 3.4.10: *When $\mathcal{K}_{Init}$ includes the reflexivity axiom,*

$$\mathcal{D}^K \models (\forall \vec{x}, \vec{y}, s).\mathbf{Knows}(\psi(\vec{y}), do(sense_\psi(\vec{y}), s)) \supset$$
$$\mathbf{Knows}(\phi(\vec{x}), do(sense_\psi(\vec{y}), s)) \equiv \mathbf{Knows}(\psi(\vec{y}) \supset \phi(\vec{x}), s),$$

$$\mathcal{D}^K \models (\forall \vec{x}, \vec{y}, s).\mathbf{Knows}(\neg\psi(\vec{y}), do(sense_\psi(\vec{y}), s)) \supset$$
$$\mathbf{Knows}(\phi(\vec{x}), do(sense_\psi(\vec{y}), s)) \equiv \mathbf{Knows}(\neg\psi(\vec{y}) \supset \phi(\vec{x}), s).$$

Proof: By Proposition 3.4.7

$$\mathcal{D}^K \models (\forall \vec{x}, \vec{y}, s).\mathbf{Knows}(W(\vec{x}), do(sense_\psi(\vec{y}), s)) \equiv$$
$$\{\psi(\vec{y}, s) \supset \mathbf{Knows}(\psi(\vec{y}) \supset W(\vec{x}), s)\} \wedge \{\neg\psi(\vec{y}, s) \supset \mathbf{Knows}(\neg\psi(\vec{y}) \supset W(\vec{x}), s)\}.$$

which is equivalent to

$$\mathcal{D}^K \models (\forall \vec{x}, \vec{y}, s).\{\, \psi(\vec{y}, s) \supset (\, \mathbf{Knows}(W(\vec{x}), do(sense_\psi(\vec{y}), s)) \equiv \mathbf{Knows}(\psi(\vec{y}) \supset W(\vec{x}), s)\,)\,\}\wedge$$

$$\{\, \neg\psi(\vec{y}, s) \supset (\, \mathbf{Knows}(W(\vec{x}), do(sense_\psi(\vec{y}), s)) \equiv \mathbf{Knows}(\neg\psi(\vec{y}) \supset W(\vec{x}), s)\,)\,\}.$$

To complete the proof it is sufficient to consider an arbitrary model of $\mathcal{D}^K$, use Corollary 3.4.9 and consider two cases: the first case is when $\psi(\vec{y}, s)$ is true in the model, the second case is when it is false. ∎

3.5 A formal comparison

Before we argue that our successor state axiom modification approach is correct with respect to Reiter's approach, let us re-write the modified successor state axiom in (3.2) into the equivalent form:

$$\begin{aligned}
&[\,a\!=\!sense(F(\vec{x}),\text{YES}) \;\supset\; (F(\vec{x},do(a,s)) \;\equiv\; true)\,] \land \\
&\qquad [\,a\!=\!sense(F(\vec{x}),\text{NO}) \;\supset\; F(\vec{x},do(a,s)) \;\equiv\; false)\,] \land \\
&\qquad\qquad [\,a\!\neq\! sense(F(\vec{x}),\text{YES}) \land a\!\neq\! sense(F(\vec{x}),\text{NO})\,] \;\supset \\
&\qquad\qquad\qquad (\,F(\vec{x},do(a,s)) \;\equiv\; \gamma_F^+(\vec{x},a,s) \;\lor\; F(\vec{x},s) \land \neg\gamma_F^-(\vec{x},a,s)\,).
\end{aligned} \tag{3.4}$$

This new form clearly demonstrates that for physical actions the usual successor state axiom (2.6) and the axiom (3.4) yield the same results, but in the case of sensing actions fluents can be set to a new value in any situation s. In the sequel, we make a *full observability* assumption: all occurrences of actions (including exogenous actions and natural actions) are observable and mentioned explicitly in a situation term.[14] In addition, we make the *uniqueness* assumption about the sequences of ground actions that we would like to consider: all occurrences of sense actions in a ground situation term are unique, i.e., we never sense the same fluent twice. Note that according to these assumptions, the second scenario in the blocks world example in Section 3.3 is impossible.

Before we formulate the correspondence between our approach and Reiter's approach to the solution of the forward projection task that involves both physical and sense actions, we define when these two tasks in $\mathcal{D}$ and $\mathcal{D}^K$ have similar structure.

Definition 3.5.1: We call the four-tuple $\{\mathcal{T}, E, \psi(s), S\}$ the representation of the forward projection task $\mathcal{T} \cup E \models \psi(S)$ if $\mathcal{T}$ is a basic action theory, E is a collection of extra axioms (about results of sense actions) added to $\mathcal{T}$, $\psi(s)$ is a regressable situation calculus formula (it has the only free variable s of the sort situation) and S is a ground situation term that can mention both physical and sense actions.

In the sequel, we use the notation $length(S)$ to denote the number of action terms that occur in S: $length(S_0) \stackrel{def}{=} 0$, $length(do(a,s)) \stackrel{def}{=} length(s) + 1$.

Definition 3.5.2: Let $\{\mathcal{D}, \emptyset, \phi, S_1\}$ be the representation of a forward projection task in $\mathcal{D}$: $\mathcal{D} \models \phi(S_1)$ and $\{\mathcal{D}^K, \Omega(S_2), \mathbf{Knows}(\phi, s), S_2\}$ be the representation of a forward projection task in $\mathcal{D}^K$: $\mathcal{D}^K \cup \Omega(S_2) \models \mathbf{Knows}(\phi, S_2)$. We call these two tasks *similar*, if the following conditions hold:

1. $\mathcal{D}$ and $\mathcal{D}^K$ are theories as described in Section 3.2.2;

[14] Recall also that there is the causal completeness assumption that the right hand side of each successor state axiom (2.6) accounts for all positive and negative effects.

2. Both S_1 and S_2 are ground situation terms and $length(S_1) = length(S_2)$;

3. For any s', s'' and actions a', a'' such that $length(s') = length(s'')$, $\mathcal{D} \models do(a', s') \sqsubseteq S_1$ and $\mathcal{D}^K \models do(a'', s'') \sqsubseteq S_2$, one of the following cases is true:

   (a) if a' is a physical action, then a'' is the same physical action, i.e., $a' = a''$,

   (b) if a' is a sense action $a' = sense(F_i(\vec{X}), V)$, then $a'' = sense_{F_i}(\vec{X})$, and vice versa. In addition, if in the term $sense(F_i(\vec{X}), V)$ the value V is YES, then
   $\Omega(do(a'', s'')) = \Omega(s'') \cup \{\mathbf{Knows}(F_i(\vec{X}), do(sense_{F_i}(\vec{X}), s''))\}$,
   otherwise, if $V =$ NO, then
   $\Omega(do(a'', s'')) = \Omega(s'') \cup \{\mathbf{Knows}(\neg F_i(\vec{X}), do(sense_{F_i}(\vec{X}), s''))\}$.

   We call actions a', a'' similar, if they satisfy all aforementioned requirements.

Informally, this definition says that tasks are similar, if the corresponding situation terms have similar structure, and whenever a sense action of one type determines that a fluent is true, a similar sense action also determines that a fluent is true, and the theory with knowledge is augmented with a formula expressing knowledge of the fluent, otherwise, if it happens that a sensed fluent is false, then the formula expressing knowledge about the negation of the fluent is added to previously constructed knowledge base of the agent.

Recall that $\mathcal{D}^K_{S_0} = \mathbf{Knows}(\kappa, S_0)$, $\mathcal{D}_{S_0} = \kappa[S_0]$, where κ is objective. The following set of statements formulate the correspondence between two projection tasks. First, we prove soundness of our approach with respect to K-based approach. Second, we prove a (partial) completeness of our approach with respect to K-based approach for a large and interesting sub-class of theories.

Theorem 3.5.3: *Let ϕ be a ground objective regressable expression*[15], *$\mathcal{D}_{una} \cup \{\kappa[S_0]\}$ decides all equality sentences, K_{init} includes the reflexivity axiom, $\{\mathcal{D}^K, \Omega(S_2), \mathbf{Knows}(\phi, s), S_2\}$ and $\{\mathcal{D}, \emptyset, \phi, S_1\}$ are representations of similar forward projection tasks in $\mathcal{D}^K$ and $\mathcal{D}$, respectively. Assume also that all successor state axioms in $\mathcal{D}_{ss}$ have the syntactic form (3.4), and assume that occurrences of all actions are observable.*

$$\textit{If } \ \mathcal{D} \models \phi(S_1), \textit{ then } \ \mathcal{D}^K \cup \Omega(S_2) \models \mathbf{Knows}(\phi, S_2)\ .$$

Proof: We prove this statement by induction on the length of situation term S_1 (we know that $length(S_1) = length(S_2)$ because projection tasks are similar).

[15] In other words, ϕ has no occurrences of quantifiers and all terms mentioned in ϕ are ground.

Base case. According to Theorems 3.4.2 and 3.4.5, we have to prove that

$$\mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \Omega^{\rho}(S_2)[S_0] \models \mathcal{R}_{\mathcal{K}}[\phi[S_2]] \quad \text{if} \qquad \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \models \mathcal{R}_{\mathcal{D}}[\phi[S_1]],$$

where $length(S_1) = length(S_2) = 0$. Because $\Omega^{\rho}(S_0) = true$, $\mathcal{R}_{\mathcal{K}}[\phi[S_0]] = \phi(S_0)$, $\mathcal{R}_{\mathcal{D}}[\phi[S_0]] = \phi(S_0)$, the statement is obviously true.

Inductive step. Let the statement be true for the case when $length(S_1) = n$. We want to prove that

$$\text{if } \ \mathcal{D} \models \phi(do(a', S_1)), \ \text{then } \ \mathcal{D}^K \cup \Omega(do(a'', S_2)) \models \textbf{Knows}(\phi, do(a'', S_2))\ ,$$

where a', a'' are similar actions. We consider two cases.

First, let a'' be a physical action $A(\vec{Y})$. Then, $\Omega(do(A(\vec{Y}), S_2)) = \Omega(S_2)$, by definition. From Proposition (3.4.6) follows that we have to prove

$$\text{if } \ \mathcal{D} \models \phi(do(A(\vec{Y}), S_1)), \text{then } \ \mathcal{D}^K \cup \Omega(S_2) \models \textbf{Knows}(\rho^1_K(\phi, A(\vec{Y})), S_2)\ ,$$

where the subscript K indicates that the 1-step regression operator ρ^1_K is defined relative to the successor state axioms in $\mathcal{D}^K$. From Lemma 2.2.10 follows that

$$\mathcal{D} \models \rho^1_D(\phi, A(\vec{Y}))[S_1] \ \text{ iff } \ \mathcal{D} \models \phi(do(A(\vec{Y}), S_1)),$$

where the subscript D indicates that the 1-step regression operator ρ^1_D is defined relative to the successor state axioms in $\mathcal{D}$. Therefore, to complete the proof it is sufficient to show that

$$\mathcal{D} \models \rho^1_K(\phi, A(\vec{Y}))[S_1] \ \text{ iff } \ \mathcal{D} \models \rho^1_D(\phi, A(\vec{Y}))[S_1]\ ,$$

because the inductive hypothesis yields that

$$\text{if } \ \mathcal{D} \models \rho^1_K(\phi, A(\vec{Y}))[S_1] \ \text{ then } \ \mathcal{D}^K \cup \Omega(S_2) \models \textbf{Knows}(\rho^1_K(\phi, A(\vec{Y})), S_2)\ .$$

From the axiom (3.4) we see that the right-hand sides of corresponding successor state axioms in $\mathcal{D}$ and $\mathcal{D}^K$, respectively, are logically equivalent for any physical action $A(\vec{Y})$. As a consequence, the definition (2.2.9) of the 1-step regression operator yields that

$$\mathcal{D} \models \rho^1_K(\phi, A(\vec{Y}))[S_1] \ \text{ iff } \ \mathcal{D} \models \rho^1_D(\phi, A(\vec{Y}))[S_1]\ ,$$

as it was required. This completes the first case.

Second, let a'' be a sense action in $\mathcal{D}^K$: $a'' = sense_{F_i}(\vec{X})$. Then, $\Omega(do(a'', S_2)) = \Omega(S_2) \cup$ **Knows**$(\pm F_i(\vec{X}), do(sense_{F_i}(\vec{X}), S_2))$, where 'plus' corresponds to the case when this sense action determines that $F_i(\vec{X})$ is true, and 'minus' – to the case when this sense action finds that $F_i(\vec{X})$ is false (i.e., 'minus' stands for negation). Because we consider similar forward projection tasks, the corresponding action term in $\mathcal{D}$ is $a' = sense(F_i(\vec{X}), \text{YES/NO})$.

Consider the single step regression operator ρ^1_D that is defined relative to the theory $\mathcal{D}$. By Lemma 2.2.10

$$\mathcal{D} \models \ \phi(do(sense(F_i(\vec{X}), \text{YES/NO}), S_1)) \ \text{ iff } \ \mathcal{D} \models \ \rho^1_D(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1].$$

Let the formula $\phi'(S_1)$ be $\phi(do(sense(F_i(\vec{X}), \text{YES/NO}), S_1))$ with all occurrences (if any) of $F_i(\vec{X}, do(sense(F_i(\vec{X}), \text{YES/NO}), S_1))$ replaced by true (false, respectively) according to the axiom (3.4) and occurrences of all other fluent literals

$$F_j(\vec{Y}, do(sense(F_i(\vec{X}), \text{YES/NO}), S_1)), \text{ where } j \neq i,$$

replaced by $F_j(\vec{Y}, S_1)$: because all successor state axioms in $\mathcal{D}$ have the syntactic form (3.4), the last sense action $sense(F_i(\vec{X}), \text{YES/NO})$ has no effect on any other fluent literal. Note that the formula $\rho^1_D(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1]$ is logically equivalent to $\phi'(S_1)$. From this observation we see that:

$$\text{if } \mathcal{D} \models \rho^1_D(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1], \text{ then } \mathcal{D} \cup \pm F_i(\vec{X}, S_1) \models \phi(S_1) .$$

Indeed, this last statement can be proved inductively over the structure of a regressable formula ϕ with the base cases when a regressable formula is an atom or equality between terms.

1. Suppose that $\phi(S_1)$ is $F_i(\vec{X}, S_1)$, and, consequently, $\phi'(S_1)$ is either true (if the result of sensing is YES) or false (if the result of sensing is NO). In the first case, our statement yields if $\mathcal{D} \models true$, then $\mathcal{D} \models F_i(\vec{X}, S_1) \supset F_i(\vec{X}, S_1)$. In the second case, the statement reduces to this: if $\mathcal{D} \models false$, then $\mathcal{D} \models \neg F_i(\vec{X}, S_1) \supset F_i(\vec{X}, S_1)$.

2. Suppose that $\phi(S_1)$ is $F_j(\vec{Y}, S_1)$, where $j \neq i$, and, consequently, $\phi'(S_1)$ is $F_j(\vec{Y}, S_1)$. No matter what is the result of sensing, our statement holds: if $\mathcal{D} \models F_j(\vec{Y}, S_1)$, then $\mathcal{D} \cup \pm F_i(\vec{X}, S_1) \models F_j(\vec{Y}, S_1)$.

3. When $\phi(S_1)$ is the negation of a fluent atom, we can use the same arguments as in the two previous steps to prove our statement.

4. If $\phi(S_1)$ is equality between terms $t_1 = t_2$, then it has no occurrences of $F_i(\vec{X}, S_1)$ and our statement obviously holds without regard to the result of sensing: if $\mathcal{D} \models t_1 = t_2$, then $\mathcal{D} \cup \pm F_i(\vec{X}, S_1) \models t_1 = t_2$.

5. Next, suppose that $\phi(S_1)$ is a disjunction of fluent literals $\bigvee_{l=1}^{m} \pm F_l(\vec{X}_l, S_1)$ with positive occurrence of the fluent atom $F_i(\vec{X}, S_1)$, and, consequently, $\phi'(S_1)$ is either the same disjunction, but without occurrences of $F_i(\vec{X}, S_1)$ (if sensing of this fluent returns NO) or true (if sensing of this fluent returns YES). If the result of sensing is NO, then our statement holds: if $\mathcal{D} \models \bigvee_{l=1, l\neq i}^{m} \pm F_l(\vec{X}_l, S_1)$, then $\mathcal{D} \models \neg F_i(\vec{X}, S_1) \supset \bigvee_{l=1}^{m} \pm F_l(\vec{X}_l, S_1)$. If the result of sensing returns YES, then our statement also holds: if $\mathcal{D} \models true$, then $\mathcal{D} \models F_i(\vec{X}, S_1) \supset \bigvee_{l=1}^{m} \pm F_l(\vec{X}_l, S_1)$. When $\phi(S_1)$ is a disjunction of fluent literals with negative occurrence of $F_i(\vec{X}, S_1)$, we can repeat the same argument with obvious modifications. When $\phi(S_1)$ is a disjunction of fluent literals with no occurrences of $F_i(\vec{X}, S_1)$, the statement obviously holds. If $\phi(S_1)$ is a disjunction of fluent literals and equalities

between terms, then we can simply repeat the same argument as above because equalities between terms have no occurrences of $F_i(\vec{X}, S_1)$.

6. Finally, without loss of generality, let $\phi(S_1)$ be a CNF $\bigwedge_{k=1}^{n} \psi_k(S_1)$, where $\psi_k(S_1)$ are prime implicates of $\phi(S_1)$. Because our statement holds for each individual prime implicate, we can conclude that it holds also in this case. Because all terms in $\phi(S_1)$ are ground (i.e., this formula has no occurrences of quantifiers), this case completes the proof.

Thus, we proved that

if $\mathcal{D} \models \phi(do(sense(F_i(\vec{X}), \text{YES/NO}), S_1))$, then $\mathcal{D} \models \pm F_i(\vec{X}, S_1) \supset \phi(S_1)$.

Therefore, by inductive hypothesis, we have that if
$\mathcal{D} \models \phi(do(sense(F_i(\vec{X}), \text{YES/NO}), S_1))$, then $\mathcal{D}^K \cup \Omega(S_2) \models \mathbf{Knows}(\pm F_i(\vec{X}) \supset \phi, S_2)$,
and, consequently, also
$\mathcal{D}^K \cup \Omega(S_2) \cup \mathbf{Knows}(\pm F_i(\vec{X}), do(sense_{F_i}(\vec{X}), S_2)) \models \mathbf{Knows}(\pm F_i(\vec{X}) \supset \phi, S_2)$.
To complete the proof, observe that by Lemma 3.4.10

if $\mathcal{D} \models \phi(do(sense(F_i(\vec{X}), \text{YES/NO}), S_1))$, then
$\mathcal{D}^K \cup \Omega(S_2) \cup \mathbf{Knows}(\pm F_i(\vec{X}), do(sense_{F_i}(\vec{X}), S_2)) \models \mathbf{Knows}(\phi, do(sense_{F_i}(\vec{X}), S_2))$.

∎

The last theorem demonstrates that if a formula about a ground situation is entailed by the theory $\mathcal{D}$ that uses modified successor state axioms, then it is also entailed by the theory $\mathcal{D}^K$ (that relies on the epistemic fluent K to represent possible worlds reasoning) augmented with axioms representing results of sensing. This theorem has important computational consequences. If one decides to employ theory $\mathcal{D}$ as a basic theory of action underlying a Golog interpreter, then evaluation of tests from Golog programs can be accomplished by checking whether a formula corresponding to a test expression is entailed from the theory $\mathcal{D}$. Because in the general case (as demonstrated by the second scenario in the example considered in Section 3.3) augmenting $\mathcal{D}^K$ with axioms representing results of sensing requires checking consistency of new axioms with the previous collection of axioms, it is computationally simpler to use $\mathcal{D}$ rather than $\mathcal{D}^K$ as an underlying basic theory of action. The theorem above provides an assurance that solving the forward projection task with respect to $\mathcal{D}$ is sound in comparison with solving the forward projection task with respect to $\mathcal{D}^K$.

However, the following example[16] demonstrates a lack of completeness, i.e., that there is

[16]This example was proposed by Yves Lespérance.

a ground formula that can be entailed from $\mathcal{D}^K$ augmented with results of sensing, but this formula is not entailed by $\mathcal{D}$.

Example 3.5.4: Consider two fluents $P(s), Q(s)$ characterized by two successor-state axioms:

$$\begin{array}{c} P(do(a,s)) \equiv \; a = sense(P, \text{YES}) \;\vee\; P(s) \wedge a \neq sense(P, \text{NO}) \\ Q(do(a,s)) \equiv \; a = sense(Q, \text{YES}) \;\vee\; Q(s) \wedge a \neq sense(Q, \text{NO}) \end{array} \quad (3.5)$$

in the theory $\mathcal{D}$ (i.e., in the theory $\mathcal{D}^K$ the fluents $P(s), Q(s)$ never change). Assume that $\mathcal{D}_{S_0} = P(S_0) \vee Q(S_0)$, i.e., $\mathcal{D}^K_{S_0} =$ **Knows**$(P \vee Q, S_0)$, and assume that all actions are always possible. Consider the situation $do(sense_P, S_0)$ that results from sensing the fluent P and let sensor report that this fluent is false. Then, it is easy to see that

$$\mathcal{D}^K \cup \textbf{Knows}(\neg P, do(sense_P, S_0)) \models \; \textbf{Knows}(P \vee Q, do(sense_P, S_0)),$$
$$\mathcal{D}^K \cup \textbf{Knows}(\neg P, do(sense_P, S_0)) \models \; \textbf{Knows}(Q, do(sense_P, S_0)),$$

but

$$\mathcal{D} \not\models \; Q(do(sense(P, \text{NO}), S_0))$$

(consider a model in which $P(S_0)$ is true, but $Q(S_0)$ is false). This happens because augmenting the initial knowledge about disjunction by the axiom about the result of sensing of one of the fluents entails in $\mathcal{D}^K$ new information about another fluent, but the theory $\mathcal{D}$ updates the truth value of the sensed fluent without providing any additional logical consequences of this update. Thus, in a general case, $\mathcal{D}$ does not get all the logical entailments of sensing results that $\mathcal{D}^K$ gets.

Later, in Chapters 5 and 6, we consider fully observable MDPs. This assumption of full observability implies that the agent is aware in what state it is; in particular, this means that the agent knows which fluents are true or false in the initial state. For this reason, we are interested in a special case of theories about S_0, when $\kappa[S_0]$ is a conjunction of ground fluent literals (if the agent's knowledge is incomplete, then $\kappa[S_0]$ does not include some of fluent literals). Logical models can assign any truth values to fluents and can represent any environment, but in the physical reality, the agent functions in a given environment and has some knowledge (it can be incomplete) about this environment. To characterize a given environment and sensing performed by the agent in the environment we have to choose which logical models can represent the environment. In particular, if we are interested only in veridical (i.e., correct) sensing actions, then we have to allow only those models which agree with results of sensing to represent the environment. Because we make the full observability assumption, all other logical models should be excluded from consideration. For example, if the situation $do(sense(P, \text{NO}), S_0)$

results from sensing that the fluent P is false in the given environment, then the logical model in which $P(S_0)$ is true should not be allowed as a faithful representation of the environment. Similarly, the sensing action $sense(P, \text{YES})$ executed in S_0 were not veridical, if it finds that the fluent P is true, but we would allow the logical model in which $P(S_0)$ is false to represent the given environment. The following formal definition is intended to capture this distinction between faithful models representing the environment and other counter-intuitive logical models that we would like to exclude. Because theories $\mathcal{D}$ and $\mathcal{D}^K$, as we described them in Section 3.2.2, have similar vocabularies, we can talk about the corresponding models of these theories: a model $\mathcal{M}_D$ of $\mathcal{D}$ and a model $\mathcal{M}_K$ of $\mathcal{D}^K$ are corresponding, if they assign the same truth values to all fluents and other common predicates.

Definition 3.5.5: Let S_1 be any ground situation term composed from actions mentioned in the theory $\mathcal{D}$ and let S_2 be a similar ground situation term composed from actions mentioned in the theory $\mathcal{D}^K$ (see Definition 3.5.2; both S_1 and S_2 can be equal to S_0). A model $\mathcal{M}_D$ of $\mathcal{D}$ is a *faithful* (representation of the environment) if for any fluent we have $\mathcal{M}_D \models \pm F_i(\vec{X}, do(sense(F_i(\vec{X}), \text{YES/NO}), S_1))$ iff $\mathcal{M}_D \models \pm F_i(\vec{X}, S_1)$, where 'plus' corresponds to the case when the sense action determines that $F_i(\vec{X})$ is true, and 'minus' – to the case when this sense action finds that $F_i(\vec{X})$ is false. Similarly, a corresponding model $\mathcal{M}_K$ of $\mathcal{D}^K$ is a *faithful* (representation of the environment) if for any fluent F_i we have $\mathcal{M}_K \models \pm F_i(\vec{X}, do(sense_{F_i}(\vec{X}), S_2))$ iff $\mathcal{M}_K \models \pm F_i(\vec{X}, S_2)$.

Because all propositions, lemmas and theorems mentioned previously are proved for arbitrary models, they remain in force when we talk only about faithful models. In the remaining part of this section, we would like to consider only those successor state axioms that have no occurrences of other fluents in the right hand side (context free axioms) [Lin and Reiter, 1997a, Reiter, 2001a]. In axioms of this form, $\gamma_F^+(\vec{x}, a)$ and $\gamma_F^-(\vec{x}, a)$ are situation independent formulae (recall that in more general case (3.2) these logical formulae can mention fluents). The basic action theories satisfying these two assumptions are still quite interesting in many domains. The following theorem states that if $\mathcal{D}$ and $\mathcal{D}^K$ are theories of this form, then solving the forward projection task in $\mathcal{D}$ is not only sound, but also complete with respect to solving the forward projection task in $\mathcal{D}^K$.

Theorem 3.5.6: *Let $\kappa[S_0]$ be a conjunction of ground fluent literals. Suppose that ϕ is a ground objective regressable expression, K_{init} includes the reflexivity axiom, $\mathcal{D}_{una} \cup \{\kappa[S_0]\}$ decides all equality sentences, $\{\mathcal{D}^K, \Omega(S_2), \mathbf{Knows}(\phi, s), S_2\}$ and $\{\mathcal{D}, \emptyset, \phi, S_1\}$ are representations of similar forward projection tasks in $\mathcal{D}^K$ and $\mathcal{D}$, respectively. Assume also that all* *successor state axioms in $\mathcal{D}_{ss}$ have the syntactic form (3.2), they are context free and assume that occurrences of all actions are observable. Then, for any faithful model $\mathcal{M}_D$ of $\mathcal{D}$ and a corresponding model $\mathcal{M}_K$ of $\mathcal{D}^K$*

$$\mathcal{M}_D \models \mathcal{D} \supset \phi(S_1) \text{ iff } \mathcal{M}_K \models \mathcal{D}^K \wedge \Omega(S_2) \supset \mathbf{Knows}(\phi, S_2).$$

Proof: We prove this statement by induction on the length of situation term S_1 (we know that $length(S_1) = length(S_2)$ because projection tasks are similar). Because occurrences of all actions are observable, we are guaranteed that this induction will cover all possible cases.

Base case is similar to the previous theorem.

Inductive step. Let the statement be true for the case when $length(S_1) = n$. We want to prove that for any faithful model $\mathcal{M}_D$ of $\mathcal{D}$ and a corresponding model $\mathcal{M}_K$ of $\mathcal{D}^K$

$$\mathcal{M}_D \models \mathcal{D} \supset \phi(do(a', S_1)) \text{ iff } \mathcal{M}_K \models \mathcal{D}^K \wedge \Omega(do(a'', S_2)) \supset \mathbf{Knows}(\phi, do(a'', S_2)),$$

where a', a'' are similar actions. We consider two cases.

First, let a'' be a physical action $A(\vec{y})$. In this case, we can repeat the same argument that we used in the previous theorem. It is easy to check that the argument from the soundness theorem works actually in both directions.

Second, let a'' be a sense action: $a'' = sense_{F_i}(\vec{X})$. Then, $\Omega(do(a'', S_2)) = \Omega(S_2) \cup$ $\mathbf{Knows}(\pm F_i(\vec{X}), do(sense_{F_i}(\vec{X}), S_2))$, where 'plus' corresponds to the case when this sense action determines that $F_i(\vec{X})$ is true, and 'minus' – to the case when this sense action finds that $F_i(\vec{X})$ is false (i.e., 'minus' stands for negation). From Lemma 3.4.10 follows that for any faithful model $\mathcal{M}_D$ of $\mathcal{D}$ and a corresponding model $\mathcal{M}_K$ of $\mathcal{D}^K$

$$\mathcal{M}_K \models \mathcal{D}^K \wedge \Omega(do(a'', S_2)) \supset \mathbf{Knows}(\phi, do(a'', S_2)) \quad \text{iff}$$
$$\mathcal{M}_K \models \mathcal{D}^K \wedge \Omega(S_2) \wedge \mathbf{Knows}(\pm F_i(\vec{X}), do(sense_{F_i}(\vec{X}), S_2)) \supset \mathbf{Knows}(\pm F_i(\vec{X}) \supset \phi, S_2).$$

According to Theorem 3.4.5, the last entailment holds iff

$$\mathcal{M}_K \models \mathcal{D}_{una} \wedge \mathcal{D}_{S_0} \wedge \Omega^{\rho}(S_2)[S_0] \supset \mathcal{R}_{\mathcal{K}}[\pm F_i(\vec{X}, S_2) \supset \phi[S_2]]$$

which is equivalent by inductive hypothesis and Theorem 3.4.2 to

$$\mathcal{M}_D \models \mathcal{D}_{una} \wedge \mathcal{D}_{S_0} \supset \mathcal{R}_{\mathcal{D}}[\pm F_i(\vec{X}, S_1) \supset \phi[S_1]],$$

Consider the single step regression operator ρ^1_D that is defined relative to the theory $\mathcal{D}$. Because by Lemma 2.2.10

$$\mathcal{D} \models \phi(do(sense(F_i(\vec{X}), \text{YES/NO}), S_1)) \quad \text{iff} \quad \mathcal{D} \models \rho^1_D(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1],$$

it remains to prove that for any faithful model $\mathcal{M}_D$ of $\mathcal{D}$

$$\mathcal{M}_D \models \mathcal{D} \supset \rho^1_D(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1] \text{ iff}$$
$$\mathcal{M}_D \models \mathcal{D}_{una} \wedge \mathcal{D}_{S_0} \supset \mathcal{R}_{\mathcal{D}}[\pm F_i(\vec{X}, S_1) \supset \phi[S_1]],$$

where ρ^1_D replaces all occurrences of F_i (if any) in the formula ϕ by true (false), depending on data returned by sensor at run time, and leaves occurrences of all remaining fluents unchanged. By Theorem 3.4.2, this last statement can be transformed to the following equivalent statement that we find more convenient to prove:

$$\mathcal{M}_D \models \rho^1_D(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1] \text{ iff } \mathcal{M}_D \models \pm F_i(\vec{X}, S_1) \supset \phi(S_1).$$

Note that this statement has two parts. In the soundness theorem, we proved that this statement is true from left to right (for any model of $\mathcal{D}$). Now we prove that this statement is true from right to left under additional assumptions stated in this theorem. Similarly to the previous theorem, we prove this last statement by induction on the structure of a regressable formula $\phi(S)$ with the base cases when this formula is an atom or equality between terms.

1. Suppose that $\phi(S_1)$ is $F_i(\vec{X}, S_1)$, and, consequently, according to the successor state axiom 3.4, $\rho^1_D(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1]$ is either true in a faithful model $\mathcal{M}_D$ of $\mathcal{D}$ (if the result of sensing is YES) or false (if the result of sensing is NO). In the first case, our statement yields if $\mathcal{M}_D \models F_i(\vec{X}, S_1) \supset F_i(\vec{X}, S_1)$, then $\mathcal{M}_D \models true$. In the second case, the statement reduces to this: if $\mathcal{M}_D \models \neg F_i(\vec{X}, S_1) \supset F_i(\vec{X}, S_1)$, then $\mathcal{M}_D \models false$, but because the model $\mathcal{M}_D$ is faithful, if a sensor reports that the fluent F_i is NO after sensing this fluent in S_1, then $F_i(\vec{X}, S_1)$ is false in $\mathcal{M}_D$.

2. Suppose that $\phi(S_1)$ is $F_j(\vec{Y}, S_1)$, where $j \neq i$, and, consequently, the left-hand side formula $\rho^1_D(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1]$ is $F_j(\vec{Y}, S_1)$. Therefore, our statement is equivalent to this: if $\mathcal{M}_D \models \pm F_i(\vec{X}, S_1) \supset F_j(\vec{Y}, S_1)$, then $\mathcal{M}_D \models F_j(\vec{Y}, S_1)$. By Theorem 3.4.2 and the definition (2.2.2) of the regression operator, this last statement can be expressed as this: if $\mathcal{M}_D \models \mathcal{D}_{una} \wedge \mathcal{D}_{S_0} \wedge \mathcal{R}_\mathcal{D}[\pm F_i(\vec{X}, S_1)] \supset \mathcal{R}_\mathcal{D}[F_j(\vec{Y}, S_1)]$, then $\mathcal{M}_D \models \mathcal{D}_{una} \wedge \mathcal{D}_{S_0} \supset \mathcal{R}_\mathcal{D}[F_j(\vec{Y}, S_1)]$. Because we assumed that all successor state axioms in $\mathcal{D}_{ss}$ are context free, the formula $\mathcal{R}_\mathcal{D}[\pm F_i(\vec{X}, S_1)]$ is logically equivalent to either $F_i(\vec{X}, S_0)$ or to $\neg F_i(\vec{X}, S_0)$, and, similarly, $\mathcal{R}_\mathcal{D}[F_j(\vec{Y}, S_1)]$ is logically equivalent to either $F_j(\vec{Y}, S_0)$ or to $\neg F_j(\vec{Y}, S_0)$ (presence or absence of negation depends on actions that occur in the situation term S_1 and on the successor state axioms). To complete the proof in this case, it is sufficient to recall that $\mathcal{D}_{S_0}$ is a conjunction of literals. If this conjunction augmented with a literal $\pm F_i(\vec{X}, S_0)$ (together with unique name axioms) entails $F_j(\vec{Y}, S_0)$ (or its negation), then $\mathcal{D}_{una} \wedge \mathcal{D}_{S_0}$ also entails the same atom $F_j(\vec{Y}, S_0)$ (or its negation, respectively) because $F_i(\vec{X}, S_0)$ and $F_j(\vec{Y}, S_0)$ are different atoms.

3. When $\phi(S_1)$ is the negation of a fluent atom, we can use the same arguments as in the two previous steps to prove our statement.

4. If $\phi(S_1)$ is equality between terms $t_1 = t_2$, then it has no occurrences of $F_i(\vec{X}, S_1)$ and our statement holds without regard to the result of sensing: $\mathcal{M}_D \models \pm F_i(\vec{X}, S_1) \supset\ t_1 = t_2$, iff $\mathcal{M}_D \models t_1 = t_2$.

5. Next, suppose that $\phi(S_1)$ is a disjunction of fluent literals $\bigvee_{l=1}^{m} \pm F_l(\vec{X}_l, S_1)$ with positive occurrence of the atom $F_i(\vec{X}, S_1)$, and, consequently, $\rho_D^1(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1]$ is either the same disjunction, but without occurrences of $F_i(\vec{X}, S_1)$ (if sensing of this fluent returns NO) or true (if sensing of this fluent returns YES). If the result of sensing is NO, then our statement is equivalent to the following: if $\mathcal{M}_D \models \neg F_i(\vec{X}, S_1) \supset (F_i(\vec{X}, S_1) \lor \bigvee_{l=1, l\neq i}^{m} \pm F_l(\vec{X}_l, S_1)\ )$, then $\mathcal{M}_D \models \bigvee_{l=1, l\neq i}^{m} \pm F_l(\vec{X}_l, S_1)$, but this last statement holds because $\mathcal{M}_D$ is a faithful model and in this model $F_i(\vec{X}, S_1)$ is false if we assume that sensing of this fluent returns the value NO. If the result of sensing returns YES, then our statement saying that if $\mathcal{M}_D \models F_i(\vec{X}, S_1) \supset (F_i(\vec{X}, S_1) \lor \bigvee_{l=1, l\neq i}^{m} \pm F_l(\vec{X}_l, S_1)\ )$, then $\mathcal{M}_D \models true$ also holds.

   When $\phi(S_1)$ is a disjunction of fluent literals with negative occurrence of $F_i(\vec{X}, S_1)$, we can repeat the same arguments with appropriate modifications.

   When $\phi(S_1)$ is a disjunction of fluent literals with no occurrences of $F_i(\vec{X}, S_1)$, the statement also holds: we can repeat arguments similar to those that we formulated in the item 2 above.

   If $\phi(S_1)$ is a disjunction of fluent literals and equalities between terms, then we can simply repeat the same argument as above because equalities between terms have no occurrences of $F_i(\vec{X}, S_1)$.

6. Finally, without loss of generality, let $\phi(S_1)$ be a CNF $\bigwedge_{k=1}^{n} \psi_k(S_1)$, where $\psi_k(S_1)$ are prime implicates of $\phi(S_1)$. Because our statement holds for each individual prime implicate, we can conclude that it holds also in this case. Because all terms in $\phi(S_1)$ are ground (i.e., this formula has no occurrences of quantifiers), this case completes the induction over structure of a regressable formula $\phi(S_1)$.

As a consequence, regression of $\phi(S_1)$ is true in a faithful model of axioms $\mathcal{D}_{una} \cup \mathcal{D}_{S_0}$ augmented by regression of the literal $\pm F_i(\vec{X}, S_1)$ if and only if $\rho_D^1(\phi, sense(F_i(\vec{X}), \text{YES/NO}))[S_1]$ is true in a faithful model of axioms $\mathcal{D}$. This completes our proof.

∎

3.6 Discussion

When an axiomatizer expresses in her axioms dynamic properties of a domain with respect to actions that her program will execute, she describes how an internal logical model of the world (truth values assigned to properties according to axioms) should evolve as a result of actions (commands that her program sends to effectors). If her program runs in the external (with respect to the program) real world, and not all dynamic aspects of the world are captured in the logical model, then real effects of actions may not coincide with effects predicted by axioms of the model. In addition, the axiomatization of the initial situation (the situation that exists before the program does any actions) will be incomplete in many realistic applications. For this reason, if the program is capable to gaining sensory feedback from the real world, it can use it to supplement the knowledge expressed in the internal model, e.g. to compare sensory readings with predicted values of properties in the model. Consequently, sensory feedback is intended to complete the gaps of unavoidable ignorance of the axiomatizer about numerous properties and causal relationships of the world (the axiomatizer is either not aware of them or not able to formalize them). Even if successor state axioms capture all relevant causal dependencies between actions and domain properties, in many realistic scenarios occurrences of exogenous actions are not observable, but can be determined indirectly from sensing; this is another reason why an agent executing a program in the real world needs sense actions.

By choosing the situation calculus as our representation language, we decide to represent properties of an application domain by fluents with a situation argument. From our perspective, fluents are internal subjective representations (‘beliefs’) of the objective real world, actions in the logical model are symbolic representations of commands that the program sends to its effectors (these actions are called physical actions) or symbolic representations of sensory inputs that the program receives from outside (these actions are called sense actions). Hence, it is convenient to treat both physical and sense actions the same in axioms.

In contrast to this perspective, a traditional approach to axiomatization of physical and sense actions separates them in different sets of axioms. A set of axioms is intended to capture the dynamics of ‘objective’ fluents with respect to causal effects of physical actions. Another set of axioms is intended to capture the dynamics of the epistemic fluent – something in the mind of an agent. It was assumed that the agent exists outside of the axiomatizer and may not be informed about certain properties of the real world that are accessible to a more knowledgeable axiomatizer. However, if the axiomatizer is supposed to express what she really knows about the world, she does her best to achieve this and she has no reason to hide valuable information from her program (that will drive actions of the agent in the real world), the first perspective seems also reasonable. Moreover, our ontology can be easily extended by adding an agent argument to all fluents (the agent 'nature' holds objective beliefs). By doing this, the axiomatizer may wish to distinguish between beliefs of agents vs. 'facts of nature', but we restrained from exploring this option in this chapter.

There are several other approaches to logical formalization of reasoning about both physical and sense actions. [De Giacomo and Levesque, 1999a] introduces guarded sensed fluent axioms (GSFA) and guarded successor state axioms (GSSA) and assumes that there is a stream of sensor readings available at any time. These readings are represented by unary sensing functions (syntactically, they look like functional fluents). [De Giacomo *et al.*, 2001] provides formal definitions of guarded theories of actions and proves soundness and correctness of an *IndiGolog* interpreter that can evaluate correctly sensor-fluent formulas if a given guarded action theory is consistent, the history of physical actions and sensing readings is coherent and a Golog program is bounded (and does not have nested search operators). As far as an implementation is concerned, both their approach and ours rely on a sort of *dynamic closed-world assumption* that is important for an efficient implementation in Prolog [Levesque and Pagnucco, 2000]. According to this assumption, in the situation resulting from execution of physical and sensing actions, the truth value of any fluent that is not known to be true with respect to an augmented theory is considered to be false. Our approach is different from the papers mentioned above because we have sensing actions rather than online sensors providing streams of data. As a consequence, our sensing actions can be mentioned explicitly in Golog programs or can be executed by the interpreter. One can imagine domains where both sensing functions and our sense actions would be useful: it remains to see how both representations can be combined together. In addition, our representation does not require consistency of sensory readings about values of fluents with the values computed from the action theory (this can be handled by solving a diagnostic task: [McIlraith, 1997]).

An interesting proposal for reasoning about perception [Pirri and Finzi, 1999, Piazzolla *et al.*, 2000] also avoids an epistemic fluent (as we do in this chapter). The authors introduce sense actions with an argument perceptible that can hold or not hold in the visual scene; this second boolean valued argument gets bound at run-time by doing the corresponding sense action in the outside world. Our representation is similar to their representation, but we allow real valued data as run-time arguments, not only boolean valued perceptibles. The major difference is that we suggest to use modified successor state axioms for reasoning about physical and sense actions in the usual language of the situation calculus, but Pirri and Finzi introduce several auxiliary predicates to reason about perception. In [Pirri and Finzi, 1999], the authors consider the important problem of mis-perception that remains outside of scope of this chapter. It would be interesting to see how this problem can be addressed using our approach to reasoning about sensing information.

[Demolombe and Pozos-Parra, 2000] proposes to consider two new sets of 'subjective fluents' BF_i and $B\neg F_i$ for each usual 'objective fluent' F_i, where the BF_i and $B\neg F_i$ are compound names of new fluents and the symbol of logical negation should be understood as a part of the names of new fluents. Informally, BF_i is understood as a belief in the fluent and $B\neg F_i$ is understood as a belief in the negation of fluent; the notations $\neg BF_i$ and $\neg B\neg F_i$ (where the external symbol of logical negation is applied to new fluents) mean the lack of belief in fluents. In addition to the standard successor state axiom for F_i, the authors propose to consider two new successor state axioms for 'subjective' fluents BF_i and $B\neg F_i$. These axioms can mention on the right hand side both physical and sensing actions, similarly to our approach.[17] However, we do not consider 'objective fluents' (in our theory they could be represented as beliefs of nature), and we consider only one kind of 'subjective fluent' that corresponds to BF_i. Other differences from our approach are that in [Demolombe and Pozos-Parra, 2000] the precondition axioms for physical actions can mention both 'subjective' and 'objective' fluents, and the sensing actions do not have an argument that gets bound at the run time. The main representational difference between our approaches stems from the fact that lack of belief is difficult to capture in our approach. The following example demonstrates this. Let $color(x, c, s)$ be a fluent that holds if the color of an object x is c in a situation s and $visible(x, s)$ be a fluent that holds if the object x is visible in s. For simplicity, we assume that a scene is static and visibility of objects does not change:

$$visible(x, do(a, s)) \equiv visible(x, s).$$

We assume that all actions are always possible and that an agent can do the action $sense(x, c)$ to find the color of x. The following successor state axiom characterizes this fluent:

$$\begin{aligned} color(x, c, do(a, s)) \equiv\ & visible(x, s) \wedge a = sense(x, c) \vee \\ & color(x, c, s) \wedge visible(x, s) \wedge \neg\exists c'(c \neq c' \wedge a = sense(x, c')). \end{aligned}$$

If in the initial situation $\neg visible(Obj, S_0)$ and $color(Obj, Green, S_0) \vee \neg color(Obj, Red, S_0)$, then in the situation S_1 that results from sensing the color of Obj we have that $\neg color(Obj, Green, S_1) \wedge \neg color(Obj, Red, S_1)$. This can be interpreted that an agent believes that the color of object

[17]Both proposals were developed independently from each other: [Demolombe and Pozos-Parra, 2000] was published in October 2000, and [Soutchanski, 2000] in August 2000.

is neither green nor red. Let us consider now an axiomatization with two subjective fluents. To represent visibility we need two fluents $believeVisible(x, s)$ and $believeNotVisible(x, s)$, they never change similar to previous axiomatization:

$$believeVisible(x, do(a, s)) \equiv believeVisible(x, s)$$
$$believeNotVisible(x, do(a, s)) \equiv believeNotVisible(x, s)$$

To represent colors of objects we also need two fluents $believeColor(x, c, do(a, s))$ and the contrary fluent $believeNotColor(x, c, do(a, s))$:

$$\begin{array}{l} believeColor(x, c, do(a, s)) \equiv believeVisible(x, s) \wedge a = sense(x, c) \vee \\ \qquad believeColor(x, c, s) \wedge believeVisible(x, s) \wedge \neg\exists c'(c \neq c' \wedge a = sense(x, c')), \\ believeNotColor(x, c, do(a, s)) \equiv believeVisible(x, s) \wedge \exists c'(c \neq c' \wedge a = sense(x, c')) \vee \\ \qquad believeNotColor(x, c, s) \wedge believeVisible(x, s) \wedge a \neq sense(x, c). \end{array}$$

Similarly, to the previous axiomatization, let $\neg believeVisible(Obj, S_0)$ and also

$$believeColor(Obj, Green, S_0) \vee believeNotColor(Obj, Red, S_0).$$

In this theory, we can easily express the lack of belief after sensing the color of Obj:

$$\begin{array}{l} \neg believeColor(Obj, Green, do(sense(Obj, Green), S_0)) \wedge \\ \qquad \neg believeNotColor(Obj, Green, do(sense(Obj, Green), S_0)), \\ \neg believeColor(Obj, Red, do(sense(Obj, Green), S_0)) \wedge \\ \qquad \neg believeNotColor(Obj, Red, do(sense(Obj, Green), S_0)). \end{array}$$

Note how these formulas express the lack of belief regarding the color of sensed object, but in the previous axiomatization the interpretation was different. In [Demolombe and Pozos-Parra, 2000], an axiomatizer must ensure consistency of beliefs: for any pair of subjective fluents $BF_i \supset \neg B\neg F_i$ (i.e., $\neg BF_i \vee \neg B\neg F_i$). If for every fluent F_i, the axiomatizer specifies also that $\neg B\neg F_i \supset BF_i$ (i.e., $BF_i \vee B\neg F_i$, the agent has one of the two beliefs), then $BF_i \equiv \neg B\neg F_i$, i.e., the axiomatizer can use only one subjective fluent to represent beliefs and lack of beliefs. Note that in Chapters 5 and 6 we make a full observability assumption when we consider MDPs. According to this assumption, in any state, the agent must have one of the two beliefs (either BF_i, i.e., the agent believes that the fluent F_i holds, or $B\neg F_i$, i.e., the agent believes that the negation of F_i holds). As a consequence, in subsequent chapters we can consider only one subjective fluent to represent beliefs.

The recent paper [Petrick and Levesque, 2002] considers combined action theories that use both a successor state axiom for the K fluent as proposed in [Scherl and Levesque, 1993] (SL theory) and successor state axioms for subjective fluents as proposed in the previously mentioned paper [Demolombe and Pozos-Parra, 2000] (DP theory). The authors formulate several additional properties that the combined action theories must satisfy (e.g., that the agent knows disjunction of fluent literals iff the agent knows one of the literals). Their combined action theories encode a correspondence between a SL theory (possible worlds) and a DP theory (‘subjective’ fluents). The authors do an in depth investigation of the correspondences between SL and DP theories and provide examples of differences between them. In particular, they prove when in combined theories the same fluent literals are known according to both SL and to DP approaches. First, this demonstrates the correctness of the DP treatment of knowledge and action in terms of K-fluent based specification. Second, this allows to translate certain SL theories into equivalent DP theories that avoid the computational drawbacks of possible world reasoning.

In this chapter we also introduce subjective fluents and propose to use both physical and sensing actions in the successor state axioms for subjective fluents. However, we investigate an issue that is different from the issues investigated in [Petrick and Levesque, 2002]. Our concern is in establishing a correspondence between the augmentation approach proposed in [Reiter, 2001a] (when sentences representing new sensory information are added to the theory to supplement incompleteness of knowledge in the initial theory) and our axiom modification approach that allows to incorporate new sensory information in an argument of run-time sensing actions.

There are several accounts of sensing in the situation calculus [Scherl and Levesque, 1993, Bacchus *et al.*, 1999, Funge, 1998, Grosskreutz, 2000, Lakemeyer, 1999, Levesque, 1996] and also [Lakemeyer and Levesque, 1998, Shapiro *et al.*, 2000] that address different aspects of reasoning about sensory and physical actions and are either inspired by the development of the epistemic fluent approach or concerned with fundamental issues of planning in the presence of incoming sensory information. Our approach is different partly because it is driven by implementational concerns. As was discussed in [Funge, 1998], the implementation of reasoning about sense actions based on $K(s', s)$ is problematic, especially in the case when the agent has to consider sensing of time-varying continuous fluents like outdoor temperature or battery voltage.

There is also an extensive literature on reasoning about sensing and physical actions in knowledge representation frameworks different from the situation calculus, e.g. [Coradeschi, 1996, De Giacomo *et al.*, 1997a, De Giacomo *et al.*, 1999, Shanahan, 1998, Doherty, 1999, Thielscher, 2000b, Coradeschi and Saffiotti, 2000, Baral and Son, 2001] to mention a few. For example, the approach to reasoning about actions advocated in [De Giacomo *et al.*, 1997a, De Giacomo *et al.*, 1999] is based on description logics; all sense actions have precondition axioms saying that an action sensing a boolean value of a proposition is possible only if both potential values of the proposition to be sensed are consistent with the current set of interpretations. A formal comparison with this and other approaches remains a topic for further research.

There are several approaches to reasoning about knowledge and action (e.g., see [Konolige, 1982, Morgenstern, 1988, Lespérance, 1991, Davis and Morgenstern, 1993a, Davis and Morgenstern, 1993b, Lespérance and Levesque, 1995]). An interesting future direction for research is reasoning about knowledge and action from the perspective of the approach developed in this chapter.

Chapter 4

Execution Monitoring

4.1 Introduction and Motivation.

Our ultimate goal is to program a robot that will be able to maintain a logical (mental) model of the world while it performs some actions in the real world.[1] This chapter will demonstrate that the mental model is useful when a program controlling the robot encounters unexpected circumstances. We assume that the robot needs to deliberate only when circumstances are such that there is no reactive procedure that may help to solve a problem. Informally speaking, we expect that our robot will not deliberate when it is simply moving towards a friend's house and the path is clear, but will start to deliberate if it finds (unexpectedly) that the friend is not at home. The usefulness of models in mobile robotics is eloquently discussed in [Thrun, 1997].

Imagine a robot that is executing a Golog program on-line, and, insofar as it is reasonable to do so, wishes to continue with this on-line program execution, no matter what exogenous events occur in the world.[2] An example of this setting, which we treat in this chapter, is a robot executing a program to build certain towers of blocks in an environment inhabited by a (sometimes) malicious agent who might arbitrarily move some block while the robot is not looking. The robot is equipped with sensors, so it can observe when the world fails to conform to its internal representation of what the world would be like in the absence of malicious agents. What could the robot do when it observes such a discrepancy between the actual world and its mental model of the world? There are (at least) three possibilities:

1. It can give up trying to complete the execution of its program.

[1] Informally speaking, values of fluents according to axioms constitute the internal mental model of an agent.

[2] We allow nondeterministic programs, so to avoid potential complications associated with an on-line execution of a program we distinguish between brave and cautious approaches to on-line execution. See Section 2.3.3 in the introductory chapter 2.

2. It can call on its programmer to give it a more sophisticated program, one that anticipates all possible discrepancies between the actual world and its internal model, and that additionally instructs it what to do to recover from such failures.

3. It can have available to it a repertoire of *general* failure recovery methods, and invoke these as needed. One such recovery technique involves planning; whenever it detects a discrepancy, the robot computes a plan that, when executed, will restore the state of the world to what it would have been had the exogenous action not occurred. Then it executes the plan, after which it resumes execution of its program. Another failure recovery method takes advantage of the nondeterminism of the program when computation in the current branch fails and it cannot be recovered using planning. For example, this may happen when actions in the current branch are delayed and if robot continues with these actions, then it will miss an important deadline at the end. This second recovery technique involves reconsidering a previous nondeterministic choice between branches of the program and taking an alternative branch. Sometimes, the robot may need an appropriate combination of two aforementioned techniques.

Execution monitoring is the robot's process of observing the world for discrepancies between "physical reality", and its "mental reality", and recovering from such perceived discrepancies especially when the plan can no longer be executed. The approach to execution monitoring that we take in this chapter is option 3 above. While option 2 certainly is valuable and important, we believe that it will be difficult to write programs that take into account all possible exceptional cases. It will be easier (especially for inexperienced programmers) to write simple programs in a language like Golog , and have a sophisticated execution monitor (written by a different, presumably more experienced programmer) keep the robot on track in its actual execution of its program.

In general, we have the following picture: The robot is executing a program on-line. By this, we mean that it is physically performing actions in sequence, as these are specified by the program. Before each execution of a primitive action and after each evaluation of a test, the execution monitor observes whether any exogenous actions have occurred. If so, the monitor determines whether the exogenous actions can affect the successful outcome of its on-line execution. If not, it simply continues with this execution. Otherwise, there is a serious discrepancy between what the robot sensed and its internal world model. Because this discrepancy will interfere with the further execution of the robot's program, the monitor needs to determine another corrected program that the robot should continue executing on-line instead of its original program. So we will understand an execution monitor as a mechanism that gets output from sensors, compares sensor measurements with its internal model and, if necessary, produces a new program whose on-line execution will make things right again.

Our purpose in this chapter is to provide a situation calculus-based account of such on-line program executions, with monitoring. To illustrate the theory and implementation, we consider two simple examples later in this chapter.

The first example is a standard blocks world as an environment in which a robot is executing a Golog program to build a suitable tower. The monitor makes use of a simple kind of planner for recovering from malicious exogenous actions performed by another agent. After the robot performs the sequence of actions generated by the recovery procedure, the discrepancy is eliminated and the robot can resume building its goal tower.

The second example is a coffee delivery robot (2.1.2) described in Section 2.1.2. In this example, temporal plans (sequences of actions with an explicit time argument) are generated from a given nondeterministic Golog program and are performed in a dynamic and uncertain environment. After doing each action, the monitor may sense the current time and find whether actions in the remaining part of the program can be scheduled according to temporal constraints in the program (including a program postcondition) and initial data. If there is no schedule for the actions remaining in the current branch of the program (i.e., one of the constraints remains violated), then the monitor (after doing auxiliary actions, if necessary) backtracks to a previous computation state and tries an alternative branch of the program. As a real-life example of this problem, consider a program that must non-deterministically perform several unrelated errands with a mobile robot in an office environment: clean a lab, deliver coffee, give a tour. If the robot was delayed in a corridor and missed the deadline of a coffee delivery request, this does not mean that the overall program failed: the robot has to execute as many of the remaining tasks as possible. The declarative framework developed here is supported by an implementation on a B21 robot manufactured by RWI. We are able to concentrate on cognitive level issues thanks to software [Burgard *et al.*, 1998, Hähnel, 1998, Hähnel *et al.*, 1998] that provides stable low level control and a reliable interface to the high level control.

Already in [McCarthy and Hayes, 1969], the authors argued that the situation calculus can serve as a basis for a high level programming language. In addition, the situation calculus has been considered as a convenient framework for the specification of constructs of programming languages, for proving properties of programs and programming languages and for transforming a given program to improve its efficiency: e.g., see [Burstall, 1969, Manna and Waldinger, 1980, Manna and Waldinger, 1981, Lin and Reiter, 1997b, Lin, 1997]. To solve a difficult problem in the software specification community, Reiter's (partial) solution [Reiter, 1991] of the frame problem [McCarthy and Hayes, 1969] is used to provide an approach for formal specification of procedures [Borgida *et al.*, 1995]; the authors propose to reify procedures and then provide axioms explaining under what circumstances a predicate or function can change from one program state to the next. Given the history of these previous research developments, the work reported in this chapter can be considered as an attempt of developing the automated tools that can assure a correct functioning of a program even in cases when the program encounters runtime deviations in the behavior of its environment. Somewhat similar research problems are considered in the area of requirements engineering [Cohen *et al.*, May 1997, Feather *et al.*, April 1998, Lamsweerde and Letier, 2000, Letier and van Lamsweerde, 2002].

The remaining part of this chapter is structured as follows. In Section 4.2, we provide general logical specifications of Golog program interpreting and execution monitoring. Two particular realizations of our framework are considered in subsequent sections. In Section 4.3, we consider planning as a recovery technique and the blocks world example. The second failure recovery technique and the coffee delivery example are described in the following Section 4.4. Finally, in Section 4.5 we formulate the correctness properties of our recovery procedures. Finally, we discuss other execution monitoring frameworks and compare them with that of proposed here.

4.2 Execution Monitoring: The General Framework

In this section we elaborate on a framework developed in our paper [De Giacomo *et al.*, 1998], without committing to any particular details of the monitor: we introduce the notion of trace and expand the framework accordingly.

In the introductory Section 2.3, we considered generation of plans from Golog programs. In particular, in Section 2.3.3 we have seen that a sequence of actions can be incrementally computed by a brave or by a cautious on-line interpreter based on a transition semantics. According to the initial Golog proposals [Levesque *et al.*, 1997, De Giacomo *et al.*, 1997b], it was assumed that once a sequence of actions is computed, it can be simply given to an execution module that has to execute each action in the real world. However, even in benign and well-structured office-type environments there is no guarantee that during the execution of actions the truth values of relational fluents or values of functional fluents computed from the successor state axioms will remain in exact correspondence to their values in the real world: other agents perform their actions independently from the robot and those actions may change the actual values of fluents. The behavior of other agents is neither predictable, nor always can be observed and often cannot be modeled an advance[3]. Nevertheless, feedback from the external world can help to account for unpredictable effects and obtain the desirable closed-loop behavior. Thus, imagine that at any stage of planning, when the interpreter selects a primitive action for execution or a test condition for evaluation, a high-level control module of the robot executes a set of predefined sense actions[4] to compare reality with its internal representation (i.e., with values of fluents according to successor state axioms). If the robot does not notice any *relevant* discrepancies, then it executes the action in reality (and proceeds executing the remaining program). Otherwise, the high-level control module (called the *monitor*) attempts to *recover* from unexpected discrepancies by finding a *corrected* program from the remaining part of the program and then proceeds with the corrected program or fails. If the remainder of a program is long enough, then it seems computationally advantageous to recover from discrepancies and then use the remaining program rather than throw it away and plan from scratch. The processes of interpreting and execution monitoring continues until the program reaches the final configuration or fails. It is natural to imagine that all discrepancies between the robot's mental world and reality are the result of sensory input, but our formal model remains the same if we assume that the high level control module is explicitly told also about exogenous actions.[5]

We want to characterize the processes of program interpretation, execution, and execution monitoring, in a declarative framework. In the sequel, we need the notion of *trace* which is a sequence of program states; intuitively, the trace for a Golog program is a history composed from states that the program goes through when it is being executed. We introduce a new sort for traces: we use letters h, h_1, h_2 and similar letters with subscripts to denote variables of sort trace. Let constant H_0 denote the empty trace, and the functional symbol $trace(\delta, h)$ denote a new trace composed from a program state δ and the previous trace h. We use a binary predicate symbol $\prec$ taking arguments of sort trace to express an order relation between traces. The notation $h_1 \preceq h_2$ is used as shorthand for $h_1 \prec h_2 \vee h_1 = h_2$. The set of traces satisfies the following axioms:

$$trace(\delta_1, h_1) = trace(\delta_2, h_2) \supset \delta_1 = \delta_2 \wedge h_1 = h_2, \tag{4.1}$$

[3]In many realistic scenarios, when the agent is placed in an unfamiliar and unmodelled environment, the agent lacks even a probabilistic model that could be used to anticipate various contingencies.

[4]As we discussed in Chapter 3, sense actions have an argument which represents data returned from sensors at the run time.

[5]In other words, for modeling purposes we can abstract away without loss of generality from the definition of how and where those exogenous (with respect to robot's mind) actions are obtained.

$$(\forall P).P(H_0) \land (\forall \delta, h)[P(h) \supset P(trace(\delta, h))] \supset (\forall h)\, P(h), \tag{4.2}$$

$$\neg(\exists \delta, h)\; trace(\delta, h) \prec H_0, \tag{4.3}$$

$$h \prec trace(\delta, h') \equiv h \preceq h'. \tag{4.4}$$

The axiom (4.2) is a second order induction axiom saying that the sort *trace* is the smallest set containing H_0 that is closed under the application of $trace$ to a *program* term and *trace*. The axiom (4.1) is a unique names axiom for traces that together with (4.2) imply that if two traces are the same, then each of them is the result of the same sequence of program states going from H_0 (informally, different sequences of program states forking from H_0 cannot join later resulting in the same trace). The axiom (4.3) means that H_0 has no predecessors, and (4.4) asserts that a trace h is a predecessor of any other trace that results from adding a program state δ to a trace h' if and only if h is a predecessor of h' or h is equal to h'. These axioms are domain independent (notice that they are similar to the foundational axioms for situations).

The on-line closed-loop system (interpreter and execution monitor) is characterized formally by a new predicate symbol *TransEM*$(s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2)$, describing a one-step transition consisting of a single $Trans(\delta_1, s_1, \delta', s')$ step of program interpretation, followed by a process, called $Monitor$, of execution monitoring. The arguments δ_1, s_1, h_1 can be understood as the current program state, the current situation, the current trace and the argument s_e represents the situation that results when several (or no) exogenous actions are executed in s_1. The role of the execution monitor is to get new sensory input in the form of sensing and exogenous actions and (if necessary) to generate a program to counter-balance any perceived discrepancy.[6] As a result of all this, the system passes from (δ_1, s_1, h_1) to the configuration (δ_2, s_2, h_2):

$$\begin{array}{l} \textit{TransEM}(s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \equiv \\ \quad (\exists \delta', s', h')\, \textit{Trans}(\delta_1, s_1, \delta', s') \land h' = trace(\delta_1, h_1) \land \\ \qquad Monitor(s_1, s_e, \delta', s', h', \delta_2, s_2, h_2). \end{array}$$

In the predicate $Monitor(s_1, s_e, \delta', s', h', \delta_2, s_n, h_2)$, the first argument s_1 is the current situation before any exogenous or sensing action happened, s_e is the situation that occurs in the result of exogenous or pre-specified sensing actions, the next three arguments δ', s', h' represent the proposed transition and the last three arguments δ_2, s_2, h_2 represent corrected values. Note that if the *Trans*-based interpreter happens to choose a branch (in a nondeterministic program

[6]We assume that sense actions are domain specific; a domain axiomatizer must specify which sequence of sense actions has to be executed after every primitive action to find its real effects.

δ_1) that leads to a dead-end, then the program may not terminate successfully even if no exogenous action will happen (compare with a *brave* version of the on-line interpreter in Section 2.3.3, see also [De Giacomo and Levesque, 1999b]). A *cautious* version avoids unsuccessful branches of δ_1:[7]

$$\begin{aligned}&\mathit{TransEM}(s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \equiv \\ &\quad (\exists\, \delta', s', h')\mathit{Trans}(\delta_1, s_1, \delta', s') \ \wedge h' = trace(\delta_1, h_1) \wedge \\ &\qquad (\exists s_g)\, Do(\delta', s', s_g) \wedge Monitor(s_1, s_e, \delta', s', h', \delta_2, s_2, h_2).\end{aligned} \tag{4.5}$$

Note that in the axiom (4.5) we use an off-line interpreter $Do(\delta', s', s_g)$ defined in 2.3.2 (see the discussion in Section 2.3.3). Note also that either $\exists a.s' = do(a, s_1)$ (a first primitive action in δ_1 is selected for execution), or $s' = s_1$ (a test is evaluated).[8]

Let e_1 be an exogenous event, which might be as simple as a sensing action or a primitive exogenous action or as complex as an arbitrary Golog program consisting of exogenous actions and domain specific sensing actions. Then, $\mathit{Trans}(e_1, s_1, e_2, s_e)$ holds if s_e is a situation resulting after doing e_1 in s_1; informally, s_e takes into account an influence of the external real world (environment). The possible configurations that can be reached with execution monitoring by a program δ given s and a trace h are those obtained by repeatedly following *TransEM* transitions interleaved with exogenous actions, i.e. those in its reflexive transitive closure *TransEM**:

$$\mathit{DoEM}(\delta, exo, s, h, s_f) \equiv \exists \delta_f, h_f.\mathit{TransEM}^*(exo, \delta, s, h, \delta_f, s_f, h_f) \wedge \mathit{Final}(\delta_f, s_f),$$

where exo is an exogenous program that interferes with the execution of the given Golog program δ; in other words, we model environment as a program exo. *TransEM** is defined as the following second-order situation calculus formula:

$$(\forall e, \delta, s, h, \delta', s', h')\ \mathit{TransEM}^*(e, \delta, s, h, \delta', s', h') \stackrel{\text{def}}{=} \forall U \forall e'[\ldots \supset U(e, \delta, s, h, e', \delta', s', h')]$$

and the ellipsis stands for the conjunction of the universal closure of the following clauses:

$$\begin{aligned}&U(e, \delta, s, h, e, \delta, s, h) \\ &U(e, \delta, s, h, e', \delta', s', h') \ \wedge\ \mathit{Trans}^*(e', s', e'', s_e) \wedge \\ &\qquad \mathit{TransEM}(s_e, \delta', s', h', \delta'', s'', h'') \supset U(e, \delta, s, h, e'', \delta'', s'', h'').\end{aligned}$$

This means that none, one, or several transitions in the execution of the exogenous program may occur between consecutive configurations of the agent program. For example, if the exogenous program is blocked in the current situation (i.e., it does not make any transitions), then the agent program is allowed to execute nonetheless.

[7]Thanks to $Do(\delta', s', s_g)$ the system is cautious enough to avoid dead-ends: the system looks ahead and does a transition from δ_1 to δ' only to a configuration from which it can terminate off-line in a goal situation s_g.

[8]$\mathit{Trans}(\phi?, s, nil, s)$ if ϕ holds in s, i.e., transitions over tests do not change the situation argument; see Section 2.3.2 for details.

We give a formal definition of the monitor and then we explain in English the intuition expressed in this definition. The behavior of a generic monitor is specified by:

$$
\begin{aligned}
&Monitor(s, s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \equiv\ \exists a. \\
&\quad [\neg Relevant(s, s_e, \delta_1, s_1, h_1) \wedge \delta_2 = \delta_1 \wedge h_2 = h_1 \wedge \\
&\qquad (s_1 = s \wedge s_2 = s_e \vee s_1 = do(a, s) \wedge s_2 = do(a, s_e)\,) \vee \\
&\quad\ Relevant(s, s_e, \delta_1, s_1, h_1) \wedge s_2 = s_e \wedge \\
&\qquad\qquad (\, s_1 = s \wedge Recover(s, \delta_1, h_1, s_e, \delta_2, h_2) \vee \\
&\qquad\qquad\quad s_1 = do(a, s) \wedge Recover(s, (a; \delta_1), h_1, s_e, \delta_2, h_2)\,)].
\end{aligned}
\tag{4.6}
$$

Here, $Relevant(s, s_e, \delta_1, s_1, h_1)$ is a predicate that specifies whether the discrepancy between s and s_e is relevant in the current configuration of the program. If this discrepancy does not matter – $\neg Relevant(s, s_e, \delta_1, s_1, h_1)$ – then the execution monitor does not modify the remaining program – $\delta_2 = \delta_1$, and keeps the current trace: $h_2 = h_1$. In addition, if the last transition was a test ($s_1 = s$), the monitor does nothing else, but if it was a primitive action ($s_1 = do(a, s)$), the monitor does the action a in the real world (this execution results in a new situation $s_2 = do(a, s_e)$). Otherwise, if this discrepancy does matter – $Relevant(s, s_e, \delta_1, s_1, h_1)$ – the monitor should recover. The predicate $Recover(s, \delta, h_1, s_e, \delta_2, h_2)$ provides for this by determining (possibly, using h_1 and δ – which is either δ_1 or $a; \delta_1$) a new program, δ_2, whose execution in situation s_e is intended to achieve an outcome *equivalent* (in a sense left open for the moment) to that of program δ, had the exogenous event not occurred and δ was simply executed in s as it was originally expected. Informally speaking, (4.6) tells the following. The monitor takes the current situation, the current program state and the trace and looks for a feedback from sensors and for information about exogenous actions. If the situation resulting from getting the feedback is such that the remaining program can be safely continued, then the monitor resumes execution of the remaining program without making any changes in the program. But if the remaining program cannot be successfully completed, then the monitor computes a new corrected program (the recovery procedure may use parts of the remaining program and the trace that contains past program states) and resumes execution with the corrected program.

A wide range of monitors can be achieved by defining $Relevant$ and $Recover$ in different ways. For example, corrective plans are suggested as a domain-independent recovery technique in our paper [De Giacomo *et al.*, 1998]. Another example is when a new program δ_2 is determined by going back to a previous program state (that involves a nondeterministic operator) and choosing an alternative branch; the past state δ_2 can be found because the trace h_1 keeps all program states which the interpreter has visited already [Soutchanski, 1999a, Soutchanski, 1999b]. As a third example, in some implemented execution monitoring systems, to recover from failures the system relies on a precompiled library of recovery plans, e.g., see [Gat, 1996, Earl and Firby, 1996, Firby, 1989].

In the next section (4.3), we consider monitoring of usual Golog programs. In the subsequent section (4.4), we adapt the framework outlined above to the case of temporal Golog programs. In both sections, we develop an implementation based on particular predicates $Relevant$ and $Recover$. Finally, in Section 4.5 we put forward constraints on $Relevant$ and $Recover$.

4.3 A Specific Monitor Based on a Planner

Now we develop a simple realization of the above general framework, by fixing on particular predicates $Relevant$ and $Recover$. This will serve as a basis for the implementation described in the section 4.3.1.

We begin by assuming that for each application domain a programmer provides:

1. The specification of all primitive actions (robot's and exogenous) and their effects, together with an axiomatization of the initial situation, as described in Section 2.1.1.

2. A Golog program γ that may or may not take into account exogenous actions occurring when the robot executes the program. Before we give a suitable definition of $Recover$, we make an important assumption about the syntactic form of the monitored program γ. We shall assume that this program has a particular form, one that takes into account the programmer's *goal* in writing it. Specifically, we assume that along with her program, the programmer provides a first order sentence describing the program's goal, or what programmers call a program *postcondition*. We assume further that this postcondition is postfixed to the program. In other words, if γ is the original program, and $goal$ is its postcondition, then the program we shall be dealing with in this section will be $\gamma\,;goal?$. This may seem a useless thing to do whenever γ is known to satisfy its postconditions, but as we shall see below, our approach to execution monitoring may *change* γ, and we shall need a guarantee that *whenever the modified program terminates, it does so in a situation satisfying the original postcondition.*

Next, we take $Relevant(s, s_e, \delta_1, s_1, h_1)$ to be the following:

$$
\begin{aligned}
&Relevant(s, s_e, \delta_1, s_1, h_1) \equiv \\
&\qquad\qquad (s_1 = s \supset \neg\exists s_g Do(\delta_1, s_e, s_g)\,) \wedge \\
&\qquad\qquad (\forall a).(s_1 = do(a, s) \supset \neg\exists s_g Do(a;\delta_1, s_e, s_g)\,).
\end{aligned} \tag{4.7}
$$

If the most recent transition is a test, we use $Do(\delta_1, s_e, s_g)$ to look ahead off-line whether the remaining program δ_1 can successfully terminate starting from the situation s_e that resulted from the exogenous or sensing actions. Similarly, we use the off-line interpreter $Do(a;\delta_1, s_e, s_g)$, if the last proposed transition was a primitive action a, to find out whether a followed by the remaining program δ_1 terminates in a goal situation s_g. Because we assume that the monitored program has a postcondition, we can simply try to execute the program off-line to see whether the postcondition still will be true (despite those interferences produced by exogenous actions which are mentioned in s_e). But if we find that there is no such situation s_g, where the program postcondition evaluates to true, then we know that the exogenous actions are relevant: we simply cannot continue with δ_1 without making any corrections. Using (4.7), the definition (4.6) of $Monitor$ becomes:

$$
\begin{aligned}
&Monitor(s, s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \;\equiv\; \exists a. \\
&\quad [\,(s_1 = s \wedge \exists s_g Do(\delta_1, s_e, s_g) \wedge \delta_2 = \delta_1 \wedge s_2 = s_e \wedge h_2 = h_1 \vee \\
&\qquad s_1 = do(a, s) \wedge \exists s_g Do(a;\delta_1, s_e, s_g) \wedge \delta_2 = \delta_1 \wedge s_2 = do(a, s_e) \wedge h_2 = h_1\,) \vee \\
&\quad (s_1 = s \supset \neg\exists s_g Do(\delta_1, s_e, s_g)\,) \wedge (\forall a).(s_1 = do(a, s) \supset \neg\exists s_g Do(a;\delta_1, s_e, s_g)\,) \wedge \\
&\qquad s_2 = s_e \wedge (\; s_1 = s \wedge Recover(s, \delta_1, h_1, s_e, \delta_2, h_2) \vee \\
&\qquad\qquad s_1 = do(a, s) \wedge Recover(s, (a;\delta_1), h_1, s_e, \delta_2, h_2)\,)].
\end{aligned}
$$

$Monitor$ takes the situation s_e reached by an exogenous program from the current situation s, and if the monitored program δ_1 terminates off-line, the monitor returns δ_1 without making any changes, else it invokes a recovery mechanism to determine a new program δ_2 from δ_1 and h_1. Therefore, $Monitor$ appeals to $Recover$ only as a last resort; it prefers to let the monitored program keep control, so long as this is guaranteed to terminate off-line in a situation where the program's goal holds. (Remember that this goal has been postfixed to the original program, as described above.)

It only remains to specify the aforementioned predicate $Recover(s, \delta, h_1, s_e, \delta_2, h_2)$ that is true whenever δ is the current state of the program being monitored[9], s is the situation prior

[9]If the most recent transition was a test, then δ is δ_1, as in the previous definition of $Monitor$, but if the last transition involved a primitive action a, then δ is $(a;\delta_1)$, where δ_1 is the remaining program.

to the occurrence of the exogenous program, s_e is the situation after the exogenous event, h_1 is the current trace, and δ_2 is a new program to be executed on-line in place of δ, beginning in situation s_e. We adopt the following specifications of $Recover$; it forms the basis of the implementation to be described later:

$$\begin{aligned} & Recover(s, \delta, h_1, s_e, \delta_2, h_2) \equiv \\ & \quad h_2 = h_1 \wedge (\ \exists p.Shortest(p, s_e, \delta) \wedge \delta_2 = (p\,;\delta) \vee \\ & \qquad \neg\exists p.Shortest(p, s_e, \delta) \wedge \delta_2 = Stop\ ), \end{aligned} \tag{4.8}$$

where

$$\begin{aligned} Shortest(p, s_e, \delta) \equiv\ & straightLineProg(p) \wedge \exists s_g.Do(p\,;\delta, s_e, s_g) \wedge \\ & [\forall p', s'.straightLineProg(p') \wedge Do(p'\,;\delta, s_e, s') \supset \\ & \qquad length(p) \leq length(p')]. \end{aligned}$$

Here, the recovery mechanism is conceptually quite simple; it determines a shortest straight-line program p such that, when prefixed onto the program δ, yields a program that terminates off-line. This is quite easy to implement; in its simplest form, simply generate all length one prefixes, test whether they yield a terminating off-line computation, then all length two prefixes, etc, until one succeeds, or some complexity bound is exceeded. If no shortest straight-line program p exists, then the recovery procedure returns the constant $Stop$ indicating that the program cannot be recovered and terminates abnormally (see the definitions (2.24) in Section 2.3.2). Notice that here we are appealing to the assumption 2 above that all monitored programs are postfixed with their goal conditions. We need something like this because the recovery mechanism *changes* the program being monitored, by adding a prefix to it. The resulting program may well terminate, but in doing so, it may behave in ways unintended by the programmer. But so long as the goal condition has been postfixed to the original program, all terminating executions of the altered program will still satisfy the programmer's intentions. The predicate $straightLineProg(p)$ and the function $length(p)$ are defined inductively. Any primitive action is a straight line program, if q is a straight line program and a is a primitive action, then $(q\,;\,a)$ is a straight line program. The length of the term a, $length(a)$, where a is a primitive action, is defined as 1, and $length(q\,;\,a)$, where q is a straight line program, is defined as $length(q) + 1$.

One disadvantage of the above recovery mechanism is that it will not recognize instances of exogenous events that happen to *help* in achieving the goal condition. In the extreme case of this, an exogenous event might create a situation that actually satisfies the goal. The above recovery procedure, being blind to such possibilities, will unthinkingly modify the current program state by prefixing to it a suitable plan, and execute the result, despite the fact that in reality, it is already where it wants to be. In effect, the recovery procedure has a built-in assumption that all exogenous events, if not neutral with respect to achieving the goal, are malicious.

We would like to make here a few comments regarding our choice of the recovery procedure. First, there are other alternative specifications, e.g., if an exogenous disturbance happens when the remainder of the program is short, it can be more advantageous to find a sequence of actions that leads directly to the goal, rather than recover from the disturbance and resume execution of the remaining program. We do not provide here alternative specifications because our intention is to introduce a conceptual framework; more sophisticated versions of recovery mechanism will be minor variations of the recovery mechanism considered here. Second, we are aware that the suggested implementation of $straightLineProg(p)$ is computationally inefficient, but it is still acceptable as long as the upper bound on the length of p is a small number. For any given application domain, computational efficiency of the implementation can be significantly improved by introducing domain specific declarative constraints on search for an appropriate $straightLineProg(p)$. It is well known that this approach leads to efficient domain specific planning [Kibler and Morris, 1981, Bacchus and Kabanza, 1996, Bacchus and Kabanza, 2000, Kvarnström and Doherty, 2001, Sierra-Santibáñez, 2001, Reiter, 2001a, Gabaldon, 2003]. Note that an efficient implementation of $straightLineProg(p)$ based on declarative constraints on the search for a shortest sequence will be applicable to the task of monitoring of an arbitrary program in the domain of application: the recovery procedure will remain the same as long as all monitored programs use primitive actions from the same basic theory of actions. Third, in the real world, if the initial situation is different from what the programmer has expected, it may happen that the original program $Prog$ cannot even be started. However, in this case we can simply run monitor once, find what kind of disturbance occurred, obtain a new (corrected) version of the program $ProgCorr$ and start executing this program. The implementation of a cautious on-line monitor that can correct initial situations is straightforward.

Finally, before we consider the implementation of the mechanism introduced in this section, we would like to formulate and prove a correctness property for our specification. Intuitively, we expect that whenever a correct closed-loop system (a Golog program coupled with an execution monitor) does a transition from (δ_1, s_1, h_1) to (δ_2, s_2, h_2) there is an execution of δ_2 leading to a situation where a postcondition $goal(s)$ is true no matter what exogenous disturbance interfered with the normal execution of the program. However, computing a new program δ_2 might not be possible in all domains. More specifically, we see from the specification (4.8) that recovering from arbitrary exogenous disturbances is possible only if an agent has a rich repertoire of primitive actions such that a program for achieving arbitrary goals can be assembled from those primitive actions. We can characterize the richness of repertoire by the condition

$$\mathcal{D} \models goal(s_g) \supset \exists p.straightLineProg(p) \wedge Do(p, s_0, s_g),$$

meaning that for any initial and goal situations there is a sequence of actions leading to a goal situation ($\mathcal{D}$ is Reiter's basic action theory that provides axiomatization of an application domain). Note that this condition holds for the blocks-world domain. Formally speaking, this condition is necessary because *TransEM* can terminate abnormally in the program state $Stop$ if recovery is impossible, but when this condition holds it guarantees that won't happen.

Theorem 4.3.1: *Let the predicate* $TransEM(s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2)$ *be defined as in (4.5), where the predicate* $Monitor(s_1, s_e, \delta', s', h', \delta_2, s_2, h_2)$ *is defined by (4.6) and its parametric predicates are defined by (4.7) and (4.8). Let* $goal(s)$ *be a program postcondition. Then the execution monitoring system satisfies the following requirement:*

$$\begin{array}{l}\mathcal{D} \models \ [goal(s_g) \supset \exists p.\ straightLineProg(p) \wedge Do(p, s_1, s_g)] \wedge \\ \qquad\qquad TransEM(s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \supset \exists s_f.Do(\delta_2, s_2, s_f) \wedge goal(s_f)\end{array}$$

Proof: To prove this statement consider two possible cases. First, let the exogenous disturbance e be irrelevant to the currently executed program, i.e. $\neg Relevant(s, s_e, \delta_1, s_1, h_1)$. Then, from the definition (4.7) follows that the right-hand side of the above implication is true. Second, let the exogenous disturbance e be relevant, i.e., $Relevant(s, s_e, \delta_1, s_1, h_1)$ is true. Then, from the definition (4.8) immediately follows that the recovered program δ_2 is such that the right-hand side of the above implication is true. ∎

4.3.1 An Implementation

In contrast to straight line or partially ordered plans, a Golog program can be arbitrary complex, including loops, recursive procedures and nondeterministic choice. For this reason, monitoring of execution of Golog programs is a more complex enterprise than monitoring execution of ordinary plans.

The above theory of execution monitoring is supported by an implementation, in Prolog, that we demonstrate here for a blocks world program. The underlying basic theory of actions for the blocks world is provided in Example 2.1.1.

Example 4.3.2: The following is a blocks world Golog program that nondeterministically builds a tower of blocks spelling "paris" or "rome". In turn, the procedure for building a Rome tower nondeterministically determines a block with the letter "e" that is clear and on the table, then nondeterministically selects a block with letter "m" and moves it onto the "e" block, etc. There is a similar procedure for *makeParis*; neither procedure has any parameters.

proc *tower* *makeParis* | *makeRome* **endProc**.
proc *makeRome*
$\pi\, b_0.[e(b_0) \wedge ontable(b_0) \wedge clear(b_0)]?\,;$
$\quad \pi\, b_1.m(b_1)?\,; move(b_1, b_0)\,;$
$\quad\quad \pi\, b_2.o(b_2)?\,; move(b_2, b_1)\,;$
$\quad\quad\quad \pi\, b_3.r(b_3)?\,; move(b_3, b_2)$
endProc
proc *makeParis*
$\pi\, b_0.[s(b_0) \wedge ontable(b_0) \wedge clear(b_0)]?\,;$
$\quad \pi\, b_1.i(b_1)?\,; move(b_1, b_0)\,;$
$\quad\quad \pi\, b_2.r(b_2)?\,; move(b_2, b_1)\,;$
$\quad\quad\quad \pi\, b_3.a(b_3)?\,; move(b_3, b_2)$
$\quad\quad\quad\quad \pi\, b_4.p(b_4)?\,; move(b_4, b_3)$
endProc

Note that this program is very simple: both procedures are sequences of non-deterministic choices of arguments followed by primitive actions. The execution of this program may easily get stuck if an action performed by another agent will make impossible one of the primitive actions that the program is supposed to execute.

We use the cautious on-line monitor of Section 4.2, and a straightforward implementation of $Monitor$ and $straightLineProg(p)$. The Prolog code is provided in Appendix A.

In our example, the initial situation is such that all blocks are on the table and clear (see Figure 4.1). There is no block with the letter "p"[10], but there are several blocks with letters for spelling "aris" and "rome", as well as blocks with letters "n" and "f" (which are irrelevant to building a tower spelling "rome" or "paris").

The program goal is:

$$Goal(s) \equiv SpellsParis(s) \vee SpellsRome(s),$$

[10] We selected this arrangement intentionally to illustrate the difference between cautious and brave interpreters.

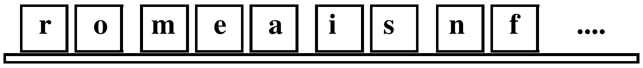

Figure 4.1: Part of the initial situation.

$$SpellsRome(s) \equiv (\exists b_0, b_1, b_2, b_3).\ R(b_3) \wedge O(b_2) \wedge M(b_1) \wedge E(b_0)\ \wedge$$
$$Ontable(b_0, s) \wedge On(b_1, b_0, s) \wedge On(b_2, b_1, s) \wedge On(b_3, b_2, s) \wedge Clear(b_3, s).$$

$$SpellsParis(s) \equiv (\exists b_0, b_1, b_2, b_3, b_4).\ P(b_4) \wedge A(b_3) \wedge R(b_2) \wedge I(b_1) \wedge S(b_0)\ \wedge$$
$$Ontable(b_0, s) \wedge On(b_1, b_0, s) \wedge On(b_2, b_1, s) \wedge On(b_3, b_2, s) \wedge On(b_4, b_3, s) \wedge Clear(b_4, s).$$

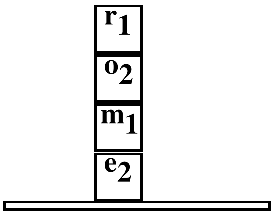

Figure 4.2: A goal arrangement of blocks.

Figure 4.2 represents an arrangement of blocks that satisfies the program goal. On that figure, r_1, o_2, m_1, e_2 are object constants that make predicates $r(X), o(X), m(X), e(X)$, respectively, true.

An Execution Trace

The original procedure $tower$ is very simple and was not designed to respond to external disturbances of any kind. However, as the trace demonstrates, the execution monitor is able to produce fairly sophisticated behavior in response to unforeseen exogenous events.

In Golog, tests do not change the situation, but all other primitive actions do. Each time the program performs a primitive action or evaluates a test, an exogenous program may occur. In the example below, any sequence of actions that is possible in the current situation can be performed by an external agent. This is realized in the implementation by interactively asking the user to provide exogenous events after each elementary program operation (test or primitive action).

The following is an annotated trace of the first several steps of our implementation for this blocks world setting. We use `this font` for the actual output of the program and *italics* for states of the Golog program $tower$ of Example 4.3.2. The symbol ":" in the Prolog implementation of the interpreter corresponds to sequential composition ";" in Golog and Prolog's term "pi" corresponds to π in Golog, etc.

```
[eclipse] onlineEM((tower : ?(goal)), s0, S).

 Program state = (nil: pi(b1,?(m(b1)) : move(b1,e1):
    pi(b2,?(o(b2)) : move(b2,b1) : pi(b3,?(r(b3)) :
        move(b3,b2))))) : ?(goal)
 Current situation: s0
```

The cautious interpreter first tried to execute $makeParis$ off-line. This failed because there is no "p" block. It then proceeded with $makeRome$.[11] According to the implementation of a cautious on-line interpreter (see section 2.3.3), $Trans$ does the first step of $makeRome$:

$\pi\, b_0.[e(b_0) \wedge ontable(b_0) \wedge clear(b_0)]?\,;$

by determining that the block e_1 will be the base of a goal tower. The remainder of the program (the "program state" in the output above) is the following:

$nil\,;$
$\pi\, b_1.m(b_1)?\,; move(b_1, e_1)\,;$
$\pi\, b_2.o(b_2)?\,; move(b_2, b_1)\,;$
$\pi\, b_3.r(b_3)?\,; move(b_3, b_2)\,;\ (goal)?$

where nil results after the first step (see axioms of $Trans$ for $\pi v.\delta$ and $\phi?$).

```
>Enter: an exogenous program or noOp if none occurs.
         move(n,m1) : move(f,n) : move(i2,o3).

 No recovery necessary. Proceeding with the next step of program.

  Program state = (nil : move(m2, e1) :
     pi(b2, ?(o(b2)) : move(b2, m2) :
         pi(b3, ?(r(b3)) : move(b3, b2)))) : ?(goal)
  Current situation: do(move(i2, o3), do(move(f, n),
                   do(move(n, m1), s0)))
```

The first exogenous program covered blocks m_1, n and o_3 (see Fig. 4.3), but the remaining program

[11] A brave interpreter would have eventually failed, without even trying $makeRome$.

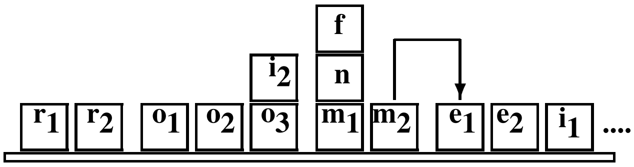

Figure 4.3: The first exogenous disturbance occurred when the program was ready to move m_2 on top of e_1.

nil ;
$move(m_2, e_1)$;
$\pi\, b_2.o(b_2)?\, ; move(b_2, m_2)$;
$\pi\, b_3.r(b_3)?\, ; move(b_3, b_2)$; $?(goal)$

can still be successfully completed because there remain enough uncovered blocks of the right kind to construct "rome", so it continues.

```
>Enter: an exogenous program or noOp if none occurs.
          move(i1,o1) : move(r2,o2).
Start recovering...

  New program = moveToTable(r2) :
     (nil : move(m2,e1) : pi(b2,?(o(b2)) :
         move(b2,m2) : pi(b3,?(r(b3)) :
             move(b3,b2)))) : ?(goal)
```

After the second exogenous program `move(i1,o1) : move(r2,o2)`, all three blocks with letter "o" are covered (see Figure 4.4).

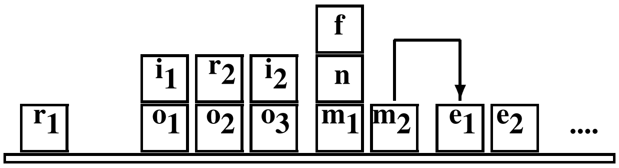

Figure 4.4: The second exogenous disturbance.

Because it is not possible to move blocks o_1, o_2, o_3 (by the precondition axiom for $move(x, y)$), the remaining program cannot be completed. Hence, the $Monitor$ gives control to $Recover$

that, with the help of the planner $straightLineProg(p)$, determines a shortest corrective sequence p of actions (namely `moveToTable(r2)`) in order to allow the program to resume, and prefixes this action to the previous program state:

$$\begin{array}{l} moveToTable(r2)\,;\,nil\ ; \\ \quad move(m_2, e_1)\,; \\ \qquad \pi\, b_2.o(b_2)?\,; move(b_2, m_2)\,; \\ \qquad\quad \pi\, b_3.r(b_3)?\,; move(b_3, b_2)\,;\ ?(goal) \end{array}$$

From this point on, the on-line evaluation continues by doing one step of the new program. If after that, no exogenous disturbances occur during the next two steps of the execution of this new program, it will reach the following program state:

$$\begin{array}{l} nil\,; \\ \quad move(o_2, m_2)\,; \\ \qquad \pi\, b_3.r(b_3)?\,; move(b_3, o_2)\,;\ ?(goal) \end{array} \tag{4.9}$$

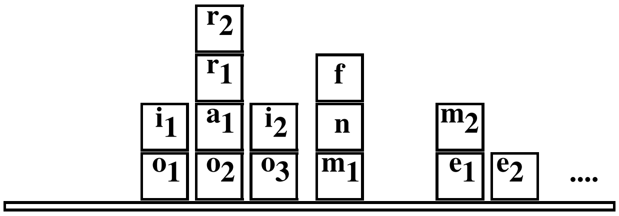

Figure 4.5: The pile of blocks covers the block o_2.

Let assume that at this point a third exogenous program occurs (see Figure 4.5):

```
>Enter: an exogenous program or noOp if none occurs.
         move(a1,o2) : move(r1,a1) : move(r2,r1).
Start recovering...
  New program = (moveToTable(r2) :
      moveToTable(r1) : moveToTable(a1)) :
         (nil : move(o2, m2) : pi(b3, ?(r(b3)) :
             move(b3, o2))) : ?(goal)

  Program state = (nil : moveToTable(r1) :
     moveToTable(a1)) : (nil : move(o2, m2) :
        pi(b3, ?(r(b3)) : move(b3, o2))) : ?(goal)
```

```
Current situation: do(moveToTable(r2),
  do(move(r2, r1), do(move(r1, a1),
    do(move(a1, o2), do(move(m2, e1),
      do(moveToTable(r2), do(move(r2, o2),
       do(move(i1, o1), do(move(i2, o3),
         do(move(f,n), do(move(n,m1), s0))))))))))
```

The recovery procedure determined that the sequence of three corrective actions

```
moveToTable(r2) : moveToTable(r1) : moveToTable(a1)
```

will lead to a situation where the program (4.9) can be resumed, and moreover, this is a shortest such correction. If no other exogenous actions happen, the program will eventually successfully terminate, having built a tower for "rome".

4.3.2 Towards a better recovery procedure

Before we introduce another implementation of the program monitor, let us consider again the program in (4.9). In a program state

$$\begin{array}{l}\pi\, b_2.o(b_2)?\,;move(b_2,m_2)\,;\\ \qquad\qquad \pi\, b_3.r(b_3)?\,;move(b_3,b_2)\,;\ ?(goal)\end{array} \tag{4.10}$$

preceding the Golog program in (4.9), to execute the test $\pi\, b_2.o(b_2)?$ the interpreter picks nondeterministically a block b_2 with the letter "o" and because at this moment only o_2 is clear and can be moved subsequently into the goal tower, it is chosen in (4.9). However, observe that after the sequence of three exogenous actions that cover o_2, this block is obviously not the best choice to move on top of m_2. Indeed, the monitor needs the sequence of three corrective actions before it can proceed, but if it would be feasible to make another choice of the block with the letter "o" imprinted on it, e.g., the block o_1, then the monitor would need only one corrective action. This observation suggests that *it can be advantageous to combine in a recovery procedure two techniques: planning and backtracking to a previous computation state*. Indeed, one can easily imagine a modified recovery procedure: try all length one sequences of actions, if none succeeds, go back to a previous computation state and try from there all length one sequences of actions, if this does not work, go to an earlier computation state and search for a single corrective action from there, etc. If all these attempts fail, try all length two sequences of actions from the current computation state, if none succeeds, go back to a previous computation state and try from there all length two sequences of actions, etc. This modified procedure will insert as few corrective actions as possible: it attempts to trade of corrective actions for backtracking to a previous program counter δ from which one can resume execution to reach a goal. Because certain primitive actions have been already executed in the real world, it is not always possible simply backtrack to a previous program state and use it as a new program when we find that the execution of the current program fails in the situation that results from exogenous actions. It may be necessary to do one or several auxiliary physical actions (and possibly 'reverse' effects of one or several already done actions), before computation can be resumed from one of the previous program states. Note that doing an auxiliary action that reverses effects of one of the previously done actions to continue execution with an alternative branch can be unwise if the alternative branch starts with the same action that just was reversed. This never happens if the monitored Golog programs are in a certain *normal form*: all branches after a nondeterministic choice start with different primitive actions. Indeed, it is always possible to transform a Golog program into a normal form by taking the initial sequence of actions (which belongs to several branches) and writing it just before the nondeterministic choice.

It is not difficult to see that this new recovery procedure is more general than the previous procedure (4.8) and that would we use this modified recovery procedure, the block o_1 were chosen in (4.10). One can imagine also even more sophisticated techniques, when one assigns rewards for achieving certain effects, costs for doing certain actions and the interpreter looks not just for a sequence of actions leading to a goal situation, but for a sequence of actions with the minimal total cost such that it provides the highest accumulated reward. However, to simplify the subsequent definitions we restrain from considering this approach.

4.3.3 An Extended Recovery Procedure.

In this section we specify a generalized recovery mechanism that was informally described in the previous section. The recovery procedure looks for a shortest sequence of corrective actions that can be prefixed to the current program state or to one of the past program states:

$$
\begin{aligned}
&Recover(s,\delta,h_1,s_e,\delta_2,h_2) \equiv \exists p.straightLineProg(p)\wedge \\
&\quad (\,\exists s_g.Do(p\,;\delta,s_e,s_g) \wedge \delta_2 = (p\,;\delta) \wedge h_2 = h_1 \vee \\
&\quad\ \exists(\delta',h').firstAlternative(h_1,\delta',h') \wedge Recover(s,(p;\delta'),h',s_e,\delta_2,h_2)\,)\wedge \\
&\quad\quad [\forall(p',s',\delta_2',h_2').straightLineProg(p') \wedge\ (\,Do(p'\,;\delta,s_e,s') \vee \\
&\quad\quad\quad \exists(\delta',h').firstAlternative(h_1,\delta',h') \wedge Recover(s,(p';\delta'),h',s_e,\delta_2',h_2')\,) \supset \\
&\qquad\qquad\qquad length(p) \leq length(p')\,],
\end{aligned}
\tag{4.11}
$$

This recovery procedure finds a shortest (possibly empty) sequence of corrective actions such that 1) either after executing them from the current program state we reach successfully the goal situation, 2) or after executing them from a past program state mentioned in the trace we can successfully compute a recovered program.[12] Because the trace keeps all program states passed by the monitored program, we need actually to consider only those past states which include alternative branches. In particular, our recovery procedure must consider the very first past program state that includes alternative nondeterministic branches. The occurrence of the predicate $Recover$ on the right hand side makes sure that program states further in the past will be recursively considered too.

The predicate $firstAlternative(h_1,\delta',h')$ holds if δ' is a first (if one looks backwards from the current trace h_1 towards H_0) program state mentioned in the trace h_1 such that δ' is a nondeterministic choice between several sub-programs. This predicate is specified by

$$
\begin{aligned}
&firstAlternative(h_1,\delta',h') \ \equiv\ h' \preceq h_1 \wedge \\
&\quad \exists(h'')h' = trace(\delta',h'') \wedge \exists(\gamma_1,\gamma_2,\gamma_3).(\delta' = (\gamma_1 \mid \gamma_2)\ \vee\ \delta' = \pi x.\gamma_3)\wedge \\
&\quad \forall(h).(h \prec h_1 \wedge \exists(h'',\delta,\gamma_1',\gamma_2',\gamma_3').h = trace(\delta,h'') \wedge (\delta = (\gamma_1' \mid \gamma_2') \vee \delta = \pi x.\gamma_3') \supset \\
&\qquad\qquad\qquad h \preceq h').
\end{aligned}
\tag{4.12}
$$

Informally, given a trace h_1, this predicate computes the sub-trace h' of h_1 such that this sub-trace is composed by a program state δ' that is a nondeterministic choice between sub-programs (i.e., δ' leads to several alternative branches) and h' is the longest sub-trace that has this property. In other words, all other sub-traces h that are composed from program states which are nondeterministic choices between sub-programs are further away in the past than the sub-trace h'.

Note that our new recovery procedure (4.11) will determine a shortest (possibly empty) sequence of actions such that execution of actions (if any) from this sequence followed by current

[12]The condition in square brackets guarantees that the sequence of corrective actions is the shortest one.

program state or followed by one of the past program states leads to a situation where the program postcondition holds. Moreover, only those past program states are considered which are nondeterministic choices between several sub-programs. Recovery using backtracking has a number of limitations. In particular, it may require 'undoing' some of the previously executed actions and this can be feasible only in domains with reversible actions.

In the next section, we consider a special but interesting case of this general recovery mechanism: when no already done actions should be undone and no corrective actions need to be inserted in front of the past program states and a new program state is determined by the recovery procedure exclusively from backtracking to a past program state.

4.4 Another Specific Monitor

We propose a domain independent execution monitoring technique for a class of high level logic-based temporal programs. The execution of a temporal program is interleaved with monitoring. Given a remaining part of a program, an execution trace, and the current time, if at this time no plan satisfying temporal constraints can be generated from the remaining program, the monitor backtracks (if possible) to a previous computation state of the restartable program where a new plan and an appropriate schedule can be constructed. We provide a declarative framework and its implementation on a mobile robot.

In this section we consider another specific monitor that implements the general specification in (4.6). Before we introduce formal definitions, we consider issues related to execution of temporal Golog programs and several simple examples.

4.4.1 Interpretation of Temporal Golog Programs

Before we proceed, let us consider again the definition of *Trans* for primitive actions with a time argument in Section 2.3.2. According to (2.23), each transition over a primitive action a determines an inequality $start(s) \leq time(a)$ saying that a may not occur in a situation s before the start time of s. Hence, an execution of a temporal program beginning in the situation S_0 may terminate in s only if a set of all such temporal inequalities together with the set of domain specific temporal constraints has a solution. Any substitution for the existentially quantified variable s obtained as a side effect of establishing the entailment $\mathcal{D} \models (\exists s) Do(\delta, S_0, s)$ specifies a unique temporal plan only if the solution is unique. Otherwise, the constructive proof that determines a situation term s will not uniquely determine occurrence times of actions in s (in a general case, a consistent set of temporal constraints may have infinitely many solutions). For this reason, when we obtain a situation term s and want to find an executable plan, we need an additional relation $Schedule(s, s')$. This relation is true if $s = s'$, or if a set of temporal inequalities determined by a sequence of actions leading from s to s' has a solution (we call it schedule) and according to the schedule actions occur as early as possible.

Examples (4.4.1) and (4.4.2) in the next section, provide motivation why we need a general framework of execution monitoring. They also show that if certain actions are significantly belated, the monitor may backtrack to old choice points in a given program and try alternative branches.

In the sequel, we describe a new specific monitor that forms the basis for the Section 4.4.5.

4.4.2 Monitoring of Temporal Golog Programs: Motivation

Let a next primitive action A be selected for the execution by the interpreter. In reality, some unexpected exogenous actions may delay completion of the previous agent's action. For this reason, before doing A that was scheduled to be executed at time T_1, the agent senses the real time T_2. If $T_2 \leq T_1$ then agent waits until T_1 and then performs A. If $T_1 < T_2$, but there is an alternative solution of the system of temporal constraints allowing the remainder of the program to be successfully rescheduled, then the execution monitor reschedules the rest of the program (in particular, it assigns new execution time $T_3 \geq T_2$ to the action A), and (after waiting for some time, if necessary) performs A at the time T_3. Otherwise, if there is no alternative schedule, then either it is too late to perform action A, or one of the subsequent actions will miss its deadline. In this case, the original plan (deduced from a given program) is doomed to fail. However, if the plan that includes action A was selected as one of possible alternatives of a nondeterministic choice, and in another branch there is a possible action B that can be scheduled at this moment, and there is a schedule for all those remaining actions which follow B, then the execution monitor abandons A (and actions that follow it) in favor of B and subsequent actions. When the monitor does this, we say that the monitor recovers from a failure by skipping belated actions and backtracking to a previous computation state. The ability to backtrack is conditioned upon the structure of a sub-program state currently being executed: only in a restartable program[13] an alternative branch can be attempted without doing

[13] A program is restartable in a current situation if it can be started anew. A related notion of a *restartable* plan has been proposed in [Beetz and McDermott, 1996]: "we call a plan p restartable if for any task t that has p as its plan, starting the execution of p, causing its evaporation, and executing p completely afterwards causes a behavior that satisfies the task t".

in the real world certain auxiliary actions or reversing physically those actions which preceded A. This loop of interpreting a program, sensing, monitoring and executing repeats until the program completes successfully or fails.

Below we consider two examples to give a motivation for our execution monitor: they use fluents and actions from the coffee delivery example (2.1.2) and use the procedure *goto*(loc,t) defined in the Golog program for a delivery robot (2.3.2).

Example 4.4.1:

This example demonstrates that backtracking to a non-deterministic choice between two sub-programs allows (sometimes) modification of a plan that would fail otherwise.

The program `visit1` relies on a functional fluent $now(s)$ (introduced in Example 2.3.2). Let the initial situation S_0 start at 0, the robot is located near the coffee machine CM, travel time from CM to $Mary$ is 10 units, travel time to Sue is 15 units, and to $Bill$ is 8 units.

proc $visit1$

$$goto(office(Mary), 1)\ ;$$
$$[goto(CM, 11)\ ; goto(office(Sue), now)\ \mid$$
$$goto(CM, 11)\ ; goto(office(Bill), now)]\ ;\ (now < 40)?$$

endProc

The robot starts to execute `visit1` in the initial situation. It has to visit $Mary$ (this takes 10 units) and then at moment 11 either return back to the coffee machine and go to the office of Sue or return back to the coffee machine and start going to the office of $Bill$. After the robot reaches one of those offices, the value of time has to be less than 40.

Let the robot select the left branch of the non-deterministic choice, i.e. it decides go to Sue. The overall plan: go to $Mary$ at 1, then come back to CM at 21 (the round trip takes 20 units), and go to Sue seems correct because the robot is supposed to arrive in the office of Sue at 36. But assume that the robot will be delayed by people walking around when it is going back to CM and it will arrive there at 26 (instead of 21); so, it is too late to start going to Sue. At this moment, the robot can change its original plan by backtracking to the non-deterministic choice and selecting another branch (go to the office of $Bill$). This plan modification does not require extra physical actions or reversing effects of those actions which have been already performed.[14] The plan modified in this way can be executed successfully, because the robot will reach $Bill$ before 40.

[14] Indeed, note that for any moment t, if $robotLocation = CM$, then $goto(CM, t)$ is equivalent to an empty sequence of actions according to the Example (2.3.2).

Example 4.4.2:

This example demonstrates that backtracking to a non-deterministic choice of an argument to a program allows (sometimes) to modify a plan that would fail otherwise.

Let us assume that the current time is 91, the robot just gave coffee to Joe, and it is located in his office. The robot has to construct and execute a plan specified by the Golog procedure `visit2`. According to `visit2`, the robot has to select a person p who does not have coffee and wants coffee during the interval from t_1 to t_2, then go to CM and pickup a coffee there (the expected travel time from Joe to CM is 10). After that, the robot has to go to the office of the selected person p, arrive there some time later and the time of the arrival must be before t_2.

proc $visit2$
$\pi\, p, t_1, t_2.\, [(\neg hasCoffee(p) \wedge wantsCoffee(p, t_1, t_2))?\ ;$
$goto(CM, 91)\ ;\ pickupCoffee(now)\ ;$
$goto(office(p), now)\ ;\ (now \leq t_2)?\ \ ].$
endProc

Let the robot select $Bill$ (he wants coffee from 100 to 110, see Example 2.1.2). The overall plan: go to the coffee machine at 91, pick up a coffee there at 101, go to the office of Bill and arrive there before 110 — seems correct, because the travel time from CM to $Bill$ is 8. But imagine that the robot was significantly delayed when it was moving from the office of Joe to CM, and at the moment when the robot has arrived there, the time was 120 units. At this moment, it is too late to start going to $Bill$, because at the very best, the robot will arrive in his office at 128 which is greater than 110. However, if the monitor backtracks to the top level non-deterministic choice and selects $Mary$ (she wants coffee from 130 to 170) instead of $Bill$, then the robot can execute a modified plan specified by `visit2`: pickup a coffee at 120, start go to $Mary$ at 120 and arrive in her office at the time 130 (the travel time to the office of $Mary$ equals 10) which is less than 170. Note that if the robot is already located at coffee machine CM, then it does not need to go anywhere given the procedure $goto(CM, 91)$; in other words, $goto(CM, 91)$ can be executed not only at 91, but at any other moment t, because $goto(CM, t)$ is equivalent to an empty sequence of actions if $robotLocation = CM$ (see the Example (2.3.2)).

4.4.3 A Specific Monitor for Temporal Golog Programs

To simplify the subsequent presentation, we consider only one primitive sense action: watching the clock and assume that there are no other exogenous actions. Thus, a high-level control module can sense in the external world only the real time. We emphasize that other information from sensors may have effects only on functioning of the low-level software (e.g., the collision avoidance program) and is not accessible to a cognitive level. We axiomatize this exogenous action by:

$\forall t, s.\ Poss(watch(t), s) \wedge time(watch(t)) = t.$

Recall that in the axiom (4.5) either $\exists a.s' = do(a, s_1)$ (a first primitive action in δ_1 is selected for execution), or $s' = s_1$ (a test is evaluated).[15] Because tests do not have temporal arguments and we consider in this section only one exogenous action $watch(t)$, there is no need to monitor evaluation of tests. Note also, that the axiom (4.5) has to be modified in the case of temporal Golog programs: we have to add the predicate $Schedule(s_1, s'')$ mentioned in Section 4.4.1, to schedule uniquely a next primitive action a selected tentatively for execution. Thus, the axiom (4.5) can be transformed as follows:

$$\begin{aligned}
&\textit{\textbf{TransEM}}(s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \equiv (\exists \delta', s', h')\ Trans(\delta_1, s_1, \delta', s') \wedge \\
&\quad h' = trace(\delta_1, h_1) \wedge \exists s''.Do(\delta', s', s'') \wedge \\
&\quad [s' = s_1 = s_2 \wedge \delta_2 = \delta' \wedge h_2 = h' \vee \\
&\quad\quad \exists a.\ s' = do(a, s_1) \wedge Schedule(s_1, s'') \wedge \\
&\quad\quad\quad Monitor(s_1, s_e, \delta', s', h', \delta_2, s_2, h_2)].
\end{aligned} \tag{4.13}$$

We also have to transform axiom (4.6) to an appropriate form that corresponds to monitoring of primitive actions only:

$$\begin{aligned}
&Monitor(s, s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \equiv (\exists t_2, a) s_e = do(watch(t_2), s) \wedge \\
&\quad [\neg Relevant(s, s_e, \delta_1, s_1, h_1) \wedge \delta_2 = \delta_1 \wedge s_1 = do(a, s) \wedge \\
&\quad\quad RunAction(a, s_e, s_2) \wedge h_2 = h_1 \vee \\
&\quad Relevant(s, s_e, \delta_1, s_1, h_1) \wedge s_2 = s_e \wedge \\
&\quad\quad Recover(s, (a; \delta_1), h_1, s_e, \delta_2, h_2)].
\end{aligned} \tag{4.14}$$

Note that sensed time may be relevant with respect to execution of δ_1 only if action a is belated: $time(a) = t_1$, $start(s_e) = t_2$, $t_2 > t_1$. If an action is belated (but the delay is not relevant), it should be rescheduled (i.e., the time argument of the action has to be replaced by a new time

[15] Recall that *Trans*$(\phi?, s, nil, s)$ if ϕ holds in s, i.e., transitions over tests do not change the situation argument.

value) because actions cannot be executed in the past. Otherwise, if an action is not belated, it is sufficient to wait until the time when the action was scheduled originally and then do the action in reality. Formally, the relation $RunAction(a, s_e, s_2)$ is specified as follows:

$$
\begin{aligned}
RunAction(a, s_e, s_2) \equiv\ & (\exists t_1, t_2, a')\ time(a) = t_1 \wedge start(s_e) = t_2 \wedge \\
& [t_2 > t_1 \wedge ReplaceTime(t_1, t_2, a, a') \wedge s_2 = do(a', s_e) \vee \\
& t_2 \leq t_1 \wedge Wait(t_2, t_1) \wedge s_2 = do(a, s_e)],
\end{aligned}
$$

where the relation $ReplaceTime(x_1, x_2, x_3, x_4)$ is true iff all occurrences of x_1 (if any) in x_3 are replaced by x_2, and x_4 is the result of this substitution; if there are no occurrences of x_1 in x_3, this relation is true if $x_4 = x_3$.[16]

The relation $Wait(t_2, t_1)$ is true iff the current clock time is t_2, and after the system waits for $t_1 - t_2$ units of time, the current clock time is t_1.

The difference between the real time $start(s_e)$ and the scheduled time $time(a)$ is relevant iff $time(a) < start(s_e)$ (action a is belated), and there is no execution of a rescheduled sequence $(a\,;\ \delta_1)$ that leads to a situation, where a post-condition of δ_1 evaluates to true:

$$
\begin{aligned}
Relevant(s, s_e, \delta_1, s_1, h_1) \equiv\ & (\exists a, t_1, t_2, a', \delta_1')\ s_1 = do(a, s) \wedge \\
& time(a) = t_1\ \wedge\ start(s_e) = t_2\ \wedge\ t_2 > t_1\ \wedge \\
& ReplaceTime(t_1, t_2, \delta_1, \delta_1') \wedge \\
& ReplaceTime(t_1, t_2, a, a') \wedge \\
& \neg\exists s'\ Do(a'; \delta_1', s_e, s').
\end{aligned}
$$

Note that we use an off-line interpreter $Do(a'; \delta_1', s_e, s')$ to look ahead whether the remaining program $a'; \delta_1'$ can terminate off-line in a goal situation. If there is no situation where this program terminates, then exogenous interferences that delayed the robot are relevant: no matter what branches in the remaining program will be chosen, the program post-condition evaluates

[16]Because primitive actions are terms that have time as their last argument $ReplaceTime$ takes as input a primitive action $a(\vec{x}, t_1)$ that the agent was supposed to execute at time t_1 and returns the action $a(\vec{x}, t_2)$ that must be executed now.

to false at the final situation. The predicate $Recover(s, (a; \delta_1), h_1, s_e, \delta_2, h_2)$ is defined by:

$$
\begin{array}{l}
Recover(s, (a; \delta_1), h_1, s_e, \delta_2, h_2) \equiv (\exists h', t_1, t_2, \delta)\ time(a) = t_1 \wedge start(s_e) = t_2 \wedge \\
\qquad (h_1 \neq H_0\ \wedge\ [\ h_1 = trace(\delta, h') \wedge \\
\qquad\qquad \exists \delta'.\, ReplaceTime(t_1, t_2, \delta, \delta')\ \wedge\ \exists a'.\, ReplaceTime(t_1, t_2, a, a') \wedge \\
\qquad\qquad\quad (\ \exists s' Do(\delta', s_e, s') \wedge \delta_2 = \delta' \wedge h_2 = h' \vee \\
\qquad\qquad\qquad \exists s' Do(a'; \delta', s_e, s') \wedge \delta_2 = (a'; \delta') \wedge h_2 = h' \vee \\
\qquad\qquad\qquad\quad Recover(s, (a; \delta_1), h', s_e, \delta_2, h_2)\ )\ ] \\
\qquad \vee\ h_1 = H_0\ \wedge\ \delta_2 = Stop \wedge h_2 = H_0).
\end{array}
$$

This predicate also appeals to the off-line executions performed by Do to look ahead whether a program will terminate in a situation where the postcondition evaluates to true. If given a non-empty trace h_1, there is a program state δ such that $h_1 = trace(\delta, h')$, and δ' – a rescheduled program δ – terminates in a goal situation (i.e., $\exists s' Do(\delta', s_e, s')$ is true), then the monitor leaves a and subsequent actions; it resumes execution from the program state δ' in situation s_e. The second disjunctive case is when $\exists s' Do(a'; \delta', s_e, s')$ is true, where a' is the appropriately rescheduled action a. In this case, the monitor executes a', but leaves subsequent actions; it resumes execution from the program state δ' in situation $do(a', s_e)$. Otherwise, recursively, it may happen that $Recover(s_1, (a; \delta_1), h', s_e, \delta_2, h_2, s_2)$ is true. If a trace h_1 is empty, then the program δ_1 cannot be recovered ($\delta_2 = Stop$), and the monitor terminates its execution. Note that we need two mutually disjoint cases when we define a recovery procedure. In the first case, the procedure abandons the action a that was scheduled for execution. In the second case, the procedure does execute a. Both cases are needed because in the temporal situation calculus fluents are processes extended in time: they are initiated and terminated by instantaneous actions. If the action a scheduled for execution happen to be an action initiating a process that will not complete in time, then the recovery procedure abandons this action a, but if a happen to be an action that terminates a delayed process, then it makes sense to execute a before trying another branch of a past program state δ'.

To formulate the correctness property of our recovery procedure, we need the following definition. The program δ (which started originally in a situation s) with the post-condition $(goal)$? is restartable in a configuration (s', δ') after any delay shorter than D if and only if

$$
\begin{array}{l}
Restartable(\delta, s', \delta', D) \stackrel{def}{=} \\
\qquad \forall s, t, s_e.[\mathit{Trans}^*(\delta, s, \delta', s') \wedge s_e = do(watch(t), s') \wedge (start(s') \leq t \leq start(s') + D)\ ] \\
\qquad\qquad \supset \exists s_f\, (Do(\delta, s_e, s_f) \wedge goal(s_f)).
\end{array}
$$

This formula can be read as follows. If (after one or several transitions) the current configuration is (s', δ'), and s_e is the situation resulting from watching the current time t in s', and time t is no later than D units from the initial time $start(s')$ of situation s', then there is a terminating execution of the original program δ from s_e such that this execution satisfies the program postcondition.

Example 4.4.3: (continuation of Example 4.4.2).
Let's reconsider the Golog program *visit2* that commands the robot go from the Joe's office at time 91 to the main office, pick up coffee there and deliver it to an employee who wants coffee at the specified interval of time. Let the situation S' be

$$S' = do(endGo(office(Joe), CM, 101), do(startGo(office(Joe), CM, 91), S_0))$$

The program *visit2* is restartable in the situation S', when the program

$$\delta' = pickupCoffee(now); goto(office(Bill), now); (now \leq 110)$$

remains to be performed, only if the delay is less than 59 time units. Indeed, $start(S') = 101$ and according to axioms of Example 2.1.2

$$wantsCoffee(p, t_1, t_2) \equiv\ p = Sue \wedge t_1 = 140 \wedge t_2 = 160 \vee\ p = Mary \wedge t_1 = 130 \wedge t_2 = 170 \vee \\ p = Bill \wedge t_1 = 100 \wedge t_2 = 110\ \vee\ p = Joe \wedge t_1 = 90 \wedge t_2 = 100.$$

Hence, for any moment of time $t > start(S') + 59$ it is too late to go to Mary because travel time to Mary from CM is 10 and the robot must arrive in her office before 170. (There are not other people who can be selected when real time $t > 160$.)

Let δ_0 be a Golog program restartable in a configuration (s_1, δ_1) after any delay shorter than D, $goal(s)$ be a situation calculus formula expressing a post-condition, S_0 be the initial situation, H_0 be an empty trace and $h_1 \neq H_0$ be a trace leading from H_0 and δ_0 to δ_1 (assume $S_0 \neq s_1$ and $\delta_0 \neq \delta_1$). Then, the recovery procedure based on backtracking is correct in the following sense:

$$\begin{array}{l}[Restartable(\delta_0, \delta_1, s_1, D) \wedge \mathit{TransEM}^*(exo, \delta_0, S_0, H_0, \delta_1, s_1, h_1) \wedge (\exists a, s)\, s_1 = do(a, s) \wedge \\ \quad (start(s) \leq t \leq start(s) + D)\ \wedge\ s_e = do(watch(t), s) \wedge \\ \quad\quad Relevant(s, s_e, \delta_1, s_1, h_1)\ ]\ \supset \\ \quad\quad\quad (\exists \delta_2, h_2)\, Recover(s, (a; \delta_1), h_1, s_e, \delta_2, h_2) \wedge \\ \quad\quad\quad\quad \exists s''\, (Do(\delta_2, s_e, s'') \wedge goal(s'')).\end{array}$$

This is a direct consequence of the definition of a restartable program and the definition of the recovery procedure. Indeed, by induction over the length of trace h_1, if this trace mentions only one program state, then from the definition of the recovery predicate it immediately follows that the original restartable program δ_0 will be determined from this trace. If this trace mentions $n+1$ program states, then using the inductive hypothesis, we see that δ_0 will also be determined from h_1. For a restartable program δ_0, the above correctness property demonstrates that even if execution of actions from this program will be delayed, then the recovery mechanism will determine a program that can be successfully executed after detecting a delay.

There are simple cases when a Golog program δ is restartable: when δ has either the syntactic form $\delta = (\gamma_1 \mid \gamma_2)$ or if $\delta = \pi x.\gamma(x)$, and at any step of execution of one of the alternative branches the rest of the branch can be skipped (if necessary) and δ can be started anew.[17] In all those cases, a program post-condition has to be flexible. For example, $goal$ has to express weaker requirement 'deliver coffee to as many people as possible', instead of overly strict requirement 'deliver coffee to all people strictly in their preferred times' (this postcondition determines termination of the recursive procedure in Example 2.3.2). If the postcondition is not flexible, then the program that uses this postcondition cannot be restarted if just one of the deliveries fails. Therefore, the proposed recovery mechanism has a number of limitations: it can recover restartable programs only and this requires postconditions of a certain syntactic form. It turns out that goals of the type 'accomplish as many errands as possible' can be conveniently expressed by adapting certain concepts from decision theory. In the next section we introduce the required fluents and predicates and then show how they can be applied to the modified version of the coffee delivery example 2.3.2.

4.4.4 Preferences and utilities.

It is convenient to view robots as economic agents that seek behavior maximizing their utility against cost. In [McFarland and Bösser, 1993], this point of view is supported by numerous examples of animals behavior. In AI there are different ways of representing utilities (e.g., see [Boutilier and Puterman, 1995, Bacchus *et al.*, 1996, Boutilier *et al.*, 1999]), but we propose here to represent them as situation calculus fluents and axiomatize accumulation of rewards by means of an appropriate successor state axiom. As we shall see, the main advantage of this approach is that the reward functions for process-oriented problems (e.g., serving requests) and for complex behaviors can be formulated without any additional representational overhead.

We represent preferences of a decision maker in an application by the fluent $Pref(s_1, s_2, s)$:

[17] In a more general case, we need a recovery mechanism that combines backtracking with inserting corrective actions as suggested in Section 4.3.3.

a situation s_1 is more preferable than a situation s_2, where both s_1 and s_2 are future situations with respect to s or one of them coincides with s. The successor state axiom for this fluent characterizes how preferences of a decision maker change between situations. As any other successor state axiom, this axiom is domain dependent. Because we want to consider the preference relation which is irreflexive and anti-symmetric, we require that in any domain this successor state axiom must satisfy the following set of constraints:

$$Pref(s_2, s_1, s) \supset \neg Pref(s_1, s_2, s),$$
$$\exists s_1, s_2.\, Pref(s_1, s_2, S_0) \wedge s_1 \neq s_2 \wedge S_0 \sqsubseteq s_1 \wedge S_0 \sqsubseteq s_2,$$
$$Pref(s_1, s_2, s) \supset (s \sqsubseteq s_1 \wedge s \sqsubseteq s_2 \wedge s_1 \neq s_2),$$

where the ordering relation $s_1 \sqsubseteq s_2$ is characterized in the set of foundational axioms (2.1)–(2.4). The fluent $Pref(s_1, s_2, s)$ characterizes preferences in a purely qualitative manner.

In our application, we express preferences using the utility fluent: $Util(v, s)$ holds when a situation s is evaluated by means a number v. We do not axiomatize real or rational numbers, but rely instead on the standard interpretation of the reals and their operations and relations. The following successor state axiom tells how the utility fluent changes:

$$\begin{array}{l} Util(v_2, do(a, s)) \equiv \\ \quad (\exists person, t, t_1, t_2, v, v_1)\,[a = giveCoffee(person, t) \wedge Util(v, s) \wedge \\ \quad\quad wantsCoffee(person, t_1, t_2) \wedge \neg hasCoffee(person, s) \wedge \\ \quad\quad\quad v_1 \leq (t_2 - t)/2 \wedge v_1 \leq t - (3t_1 - t_2)/2 \wedge v_2 = v + v_1\,] \vee \\ \quad Util(v_2, s) \wedge \neg(\exists person, t) a = giveCoffee(person, t) \end{array}$$

The utility fluent is intended to represent a cumulative reward gained for a certain behavior. In the case of our example, utility remains the same if the robot executed an action different from $giveCoffee$. Otherwise, the new value of the utility v_2 is a sum of the utility v in the previous situation and a reward v_1 for doing the action $giveCoffee(person, t)$ at time t. The reward v_1 is bounded from above by a pair of linear functions: one of them is increasing with time, the other is decreasing with time. Note that v_1 has a maximal value $3t_1$ if $t = t_1$ (this is a moment of time when the linear functions intersect) and $v_1 \leq 0$ if either $t \leq (3t_1 - t_2)/2$ or $t \geq t_2$. Linear functions are chosen intentionally to provide the maximum reward at the moment t_1: this reward function encourages the robot to arrive near the office of an employee just before t_1 (and wait for some time, if the robot arrives too early). Axioms P in conjunction with the following axiom say that preferences can be expressed in terms of utilities:

$$\begin{array}{l} Pref(s_2, s_1, s) \equiv \exists v_2, v_1\, (s \sqsubseteq s_1 \wedge s \sqsubseteq s_2 \wedge Util(v_2, s_2) \wedge Max(v_2, s_2) \wedge \\ \quad\quad\quad Util(v_1, s_1) \wedge Max(v_1, s_1) \wedge v_2 > v_1), \end{array}$$

where $Max(v, s)$ holds if v is a maximal value with respect to the set of those linear temporal inequalities which constrain values of time variables mentioned in the situational term s (as a positive side effect, this maximization operation allows us to choose numerical values for all temporal variables mentioned in the situation terms). Therefore, in the intended interpretation, the value v can be determined by solving the corresponding linear programming problem.

In the section (2.3.1), any solution of the entailment task $\mathcal{D} \models (\exists s)Do(\delta, S_0, s)$, i.e., any substitution for s, has been considered an appropriate sequence of actions. But if a decision maker has preferences over different hypothetical plans, then they all have to be compared with each other and only the best plan must be taken as a solution of the planning problem. Thus, we need the predicate $BestTrans(\delta, s, \delta', s')$ saying that the configuration (δ', s') is the best one that can be reached from the configuration (δ, s) by doing one step of computation along a sequence of actions that constitutes an optimal plan:

$$\begin{array}{l} BestTrans(\delta, s, \delta', s') \equiv \\ \quad (\exists s_f) Trans(\delta, s, \delta', s') \wedge \boldsymbol{Do}(\delta', s', s_f) \wedge Pref(s_f, s, s) \wedge \\ \qquad \neg(\exists s_1)\, (\boldsymbol{Do}(\delta, s, s_1) \wedge Pref(s_1, s_f, s)). \end{array}$$

We have now a convenient framework to represent the requirement ‘deliver coffee to as many people as possible’.

First, we introduce the predicate $goal(s)$ (assume $goal(S_0)$ is true):

$$\begin{array}{l} goal(do(a, s)) \equiv \\ \quad (\exists p, t)\, a = giveCoffee(p, t) \wedge Pref(do(a, s), s, S_0) \vee \\ \qquad \neg(\exists p, t)\, a = giveCoffee(p, t) \wedge goal(s) \end{array} \tag{4.15}$$

Informally, a final situation must be such that every $giveCoffee$ action results in a better situation. According to the utility function, the robot shall never give a coffee to a person p after the time t_2 if $wantsCoffee(p, t_1, t_2)$. Hence, the test $(goal)?$ evaluates to true iff utility either increases monotonically or remains the same along the sequence of actions leading to the final situation where this test is evaluated. In the sequel, we consider a modified coffee delivery program γ (the Golog code of this program is provided in the next section). The interpreter coupled with the monitor generate and schedule plans (and later modify and re-schedule plans, if necessary) using the program $\delta = \gamma; (goal)?$, i.e., the test $(goal)?$ is appended to γ to represent the post-condition (4.15).

Second, because we want to compute optimal schedules in our implementation and reschedule remaining actions in an optimal way, we have to transform the expression characterizing transitions of a closed-loop system between configurations. Thus, the axiom (4.13) has to be replaced by the following axiom:

$$\begin{aligned}
&TransEM(s_e, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \equiv (\exists \delta', s', h') \\
&\ BestTrans(\delta_1, s_1, \delta', s') \land h' = trace(\delta_1, h_1) \land \\
&\ [\, s' = s_1 = s_2 \land \delta_2 = \delta' \land h_2 = h' \lor \\
&\ \ (\exists a) s' = do(a, s_1) \land Monitor(s_1, s_e, \delta', s', h', \delta_2, s_2, h_2)].
\end{aligned} \tag{4.16}$$

The definition

$\textit{DoEM}(\delta, exo, s, h, s_f) \equiv \exists \delta_f, h_f . \textit{TransEM}^*(exo, \delta, s, h, \delta_f, s_f, h_f) \land \textit{Final}(\delta_f, s_f)$,

where *TransEM** is the reflexive transitive closure of *TransEM* defined in (4.16) remains the same. Note that the program exo here consists exclusively of actions sensing the current time as mentioned in the beginning of Section 4.4.3. Everything else in Section 4.4.3 also remains in force. Obviously, given a situation s, any substitution for an existentially quantified variable s_f obtained as a solution of the entailment task $\mathcal{D} \models (\exists s_f) DoEM(\delta, s, s_f)$ corresponds to an optimal plan 'deliver coffee to as many people as possible'. The optimal plan is computed (and re-computed) on-line taking into account real measurements of time.

4.4.5 An Implementation on a Robot

The interpreter and monitor are being used to control an autonomous robot Golem to perform temporal scheduling tasks in an office environment (see Section 2.5).

It is important to emphasize that all sensor readings performed by low-level software are not accessible to the high-level control. The execution monitor described in the section (4.4.3) uses Unix internal clock as a sole sensor input. This is the limitation of the current experimental version, not the limitation of our general framework.

The interpreter for temporal Golog based on transitional semantics with an execution monitor has been implemented in Eclipse 3.5.2 Prolog. The ECRC Common Logic Programming System Eclipse 3.5.2 provides a built-in Simplex algorithm for solving linear equations and inequalities over the reals. It provides also a built-in predicate *rmax(T)* that maximizes the value that variable *T* may have with respect to the current set of temporal constraints.

The relations $DoEM$, $TransEM$, $Monitor$, $Relevant$, $Recover$, lead to a natural implementation in Prolog of an execution monitor for arbitrary restartable temporal Golog programs. An implementation of the $Trans$-based interpreter for temporal Golog programs is available in Appendix B.1. An implementation of the monitor is available in Appendix B.2. The source code of the example considered later in this section can be found in Appendix B.4.

In our experiments, we run the following program.

proc $serveCoffee(t)$
$(\ noCoffeeDeliveryIsPossible\ \mid$
if $holdingCoffee$
then $serveOneCoffee(t)$
else $goto(CM, t);\ pickupCoffee(now);$
$serveOneCoffee(now)\ ).$
endProc

proc $noCoffeeDeliveryIsPossible$
$[(\neg\exists p, t', t'', wait).$
$wantsCoffee(p, t', t'') \wedge \neg hasCoffee(p) \wedge wait \geq 0 \wedge$
$t' \leq now + wait + travelTime(robotLocation, CM) + travelTime(CM, office(p)) \leq t'']?$
endProc

proc $serveOneCoffee(t)$
$(\pi p, t_1, t_2, wait)\ [\ \{wantsCoffee(p, t_1, t_2) \wedge$
$\neg hasCoffee(p)\ \wedge\ wait \geq 0\ \wedge$
$t_1 \leq t + wait + travelTime(robotLocation, office(p)) \leq t_2\}?\ ;$
$goto(office(p), t + wait)\ ;$
$giveCoffee(p, now)\ ;$
$serveCoffee(now)\ ]$
endProc

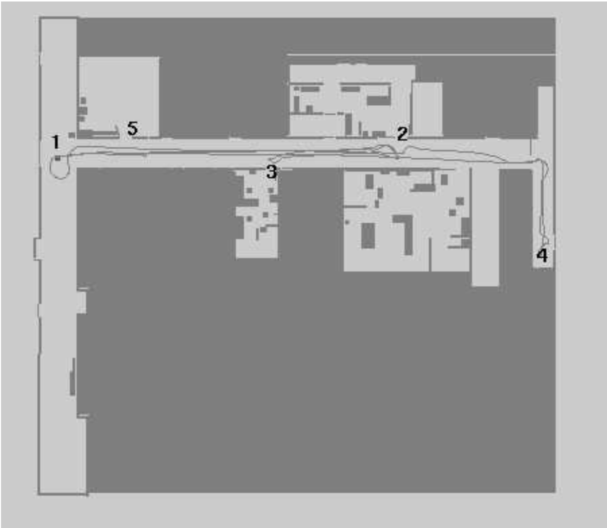

Figure 4.6: Locations of offices on the map: 1) park, 2) coffee machine, 3) Yves, 4) Ray, 5) Sam

In our experiments, we used the following data:

$$\begin{aligned} wantsCoffee(p, t_1, t_2) \equiv\ & p = Sam \wedge t_1 = 600 \wedge t_2 = 700 \vee \\ & p = Ray \wedge t_1 = 360 \wedge t_2 = 440 \vee \\ & p = Yves \wedge t_1 = 160 \wedge t_2 = 240. \end{aligned}$$

Robot travel times:

$$\begin{aligned} & travelTime(CM, office(Yves)) = 45, \\ & travelTime(CM, office(Ray)) = 120, \\ & travelTime(CM, office(Sam)) = 75, \\ & travelTime(Park, CM) = 100, \\ & travelTime(office(Yves), office(Ray)) = 120, \\ & travelTime(office(Ray), office(Sam)) = 150. \end{aligned}$$

If we run the program $\delta = serveCoffee(20); (goal)?$ off-line (i.e., if we solve the entailment problem $\mathcal{D} \models (\exists s) Do(\delta, S_0, s)$ to find a substitution for s) we can find the optimal plan

$$
\begin{aligned}
S = {} & do(giveCoffee(Sam, 600), do(endGo(CM, office(Sam), 600), \\
& do(startGo(CM, office(Sam), 525), do(pickupCoffee(480), \\
& do(endGo(office(Ray), CM, 480), do(startGo(office(Ray), CM, 360), \\
& do(giveCoffee(Ray, 360), do(endGo(CM, office(Ray), 360), \\
& do(startGo(CM, office(Ray), 240), do(pickupCoffee(210), \\
& do(endGo(office(Yves), CM, 210), do(startGo(office(Yves), CM, 165), \\
& do(giveCoffee(Yves, 165), do(endGo(CM, office(Yves), 165), \\
& do(startGo(CM, office(Yves), 120), do(pickupCoffee(120), \\
& do(endGo(Park, CM, 120), \\
& do(startGo(Park, CM, 20), s0))))))))))))))))),
\end{aligned}
$$

that has the utility 127.5. According to the optimal plan, the robot has to deliver coffee first to $Yves$ (at 165), then to Ray (at 360), and finally to Sam (at 600). Note that the procedure $serveOneCoffee$ is restartable in the computation state when the robot has arrived near the office (before it gives coffee). To demonstrate that our recovery procedure works as intended we did the following experiment. When the robot was moving from the coffee machine (the location 2 on the map) to the office of $Yves$ (the location 3 on the map), we intentionally delayed the robot. As a result, it has arrived at 260, much later than it was supposed to arrive according to a schedule. At this time, were the robot executed the sequence of actions

$$endGo(office(Yves), 260); giveCoffee(Yves, 260),$$

the utility would decrease and as a consequence the post-condition $(goal)$? would be false. The predicate $Relevant$ detects this potential violation using an off-line simulation of the remaining program (the predicate Do in the definition of $Relevant$ evaluates to false if an off-line execution of δ fails). Hence, the monitor attempts to recover from the failure in the optimal plan. The predicate $Recover$ backtracks recursively to a past computation state from which a new optimal plan can be computed: this state is the top most non-deterministic choice of person to whom a coffee has to be delivered now (see the procedure $serveOneCoffee$). As we observed, the robot executed the action $endGo(office(Yves), 260)$, and then (after some waiting) started to move from the office of Yves to the office of Ray without giving coffee to Yves (because the robot holds coffee it goes straight to Ray's office without picking up another coffee). In the set of other experiments, we have seen that the closed-loop system (the incremental interpreter coupled with the monitor) behaved exactly as intended.

4.5 Correctness Properties for Execution Monitors

With definitions for $TransEM$ and $Monitor$, as in Section 4.2, it becomes possible to formulate, and ultimately prove, various correctness properties for execution monitors. These properties are intended to capture suitable concepts of *controllability* following the intuition behind similar concepts introduced for discrete event systems [Ramadge and Wonham, 1989]. Informally, controllability is the property that characterizes a closed-loop system (a Golog program coupled with the execution monitor): this is the ability of a monitored program to behave correctly even if exogenous actions occur during the robot's execution of the program. There are many possible definitions, with varying degrees of generality, of what counts as correct behavior of a monitored system. We focus here on various correctness properties one might want to prove of the monitor.

Recall that the general execution monitor, as specified by (4.6), is the 8-argument relation $Monitor(s_c, s_e, \delta, s, h, \delta', s', h')$, meaning that whenever δ is the state of monitored program in situation s with the trace of past program states h, and s_e is a situation resulting from an exogenous event occurrence at the current situation s_c, then the on-line execution should resume in s' with the new program δ'. Then, by analogy with Floyd-Hoare style correctness and termination properties, we can formulate a variety of verification tasks, some examples of which we now describe. These are parameterized by two predicates:

1. $P(s)$, a desirable property that a terminating situation s must satisfy. For example, P might describe a postcondition of the program being monitored.

2. $Q(\delta, h, s_c, s_e)$, a relationship between the current program state δ, the current trace h, the current situation s_c, and s_e, the situation resulting from an exogenous event occurring in s_c. For example, Q might express that δ terminates off-line both when executed in s_c and also when executed in s_e. As another example, we can consider the formula $Q(\delta, h, s_c, s_e)$ saying that for any pair s_c, s_e there exists a straight-line program such that the execution of this program in s_e results in some situation s_n and the execution of δ from s_n terminates.

Weak Termination and Correctness

$$(\forall \delta, h, s_c, s_e).Q(\delta, h, s_c, s_e) \supset$$
$$Monitor(s_c, s_e, \delta, s, h, \delta', s', h') \supset (\exists s'').Do(\delta', s', s'') \wedge P(s'').$$

The task here is to verify that, under condition Q, whenever $Monitor$ determines a new program with which to resume the system computation after an exogenous event occurrence, that program has a terminating (off-line) computation resulting in a final situation in which P holds.

Strong Termination and Correctness

$$(\forall \delta, s_c, s_e).Q(\delta, h, s_c, s_e) \supset (\exists \delta', s', h').Monitor(s_c, s_e, \delta, s, h, \delta', s', h') \ \wedge$$
$$(\exists s'').Do(\delta', s', s'') \wedge P(s''),$$

Under condition Q, $Monitor$ always determines a new program with which to resume the system computation after an exogenous event occurrence, and that program has a terminating (off-line) computation resulting in a final situation in which P holds.

Even Stronger Termination and Correctness

$$(\forall \delta, h, s_c, s_e).Q(\delta, s_c, s_e) \supset$$
$$(\exists \delta', s', h').Monitor(s_c, s_e, \delta, s, h, \delta', s', h') \wedge (\exists s'').Do(\delta', s', s'') \ \wedge$$
$$(\forall s'').Do(\delta', s', s'') \supset P(s'').$$

Here, the correctness property is that under condition Q, $Monitor$ always determines a new program that terminates off-line, and all these terminating situations satisfy P.

It is also possible to formulate various correctness properties for the entire monitored system (the agent together with its environment), for example, the weak property that provided the monitored program terminates, then it does so in a desirable situation:

$$(\forall \delta, s_f, exo).DoEM(\delta, exo, S_0, H_0, s_f) \supset P(s_f).$$

Other variations on the above themes are possible, but our purpose here is not to pursue these issues in depth, but simply to point out that correctness properties for monitored systems are easily formulated within our framework. Moreover, because this framework is entirely within the situation calculus, such correctness proofs can be constructed totally within a classical predicate logic.

4.6 Discussion

There are several systems designed to interleave monitoring with plan execution, e.g.: FLEA [Cohen *et al.*, May 1997, Feather *et al.*, April 1998], IPEM [Ambrose-Ingerson and Steel, 1988], PLANEX [Fikes *et al.*, 1972], PRS [Georgeff and Lansky, 1987, Georgeff and Ingrand, 1989], ROGUE [Haigh and Veloso, 1996], SIPE [Wilkins, 1988], SOAR [Laird and Rosenbloom, 1990], SPEEDY [Bastié and Régnier, 1996], XII [Golden *et al.*, February 1996, Golden and Weld, 1996], the robot control system described in [Nourbakhsh, 1997]. We differ from these and similar proposals, first by using the very expressive classical predicate logic language for specifying application domains with temporal constraints, secondly by the fact that ours is a story for monitoring arbitrary *programs*, not simply straight line or partially ordered plans. Moreover, we do not assume that the monitored plan is generated automatically from scratch, but rather that it has been computed from a high-level Golog program provided by a programmer.

The idea of developing declarative theories to design controllers for hybrid systems can be traced back to [Kohn, 1988, Kohn, 1991]; the subsequent developments are described in [Kohn and Nerode, 1993b, Kohn and Nerode, 1993a].

In a sequence of papers [Schoppers, 1987, Schoppers, 1989, Schoppers, 1992] Schoppers proposes and defends the idea of "universal plans", which "address the tension between reasoned behavior and timely response by caching reactions for classes of possible situations". In particular, in [Schoppers, 1989], he writes that "this technique reduces the average time required to select a response at the expense of the space required to store the cache – the classic time-space trade-off" and then argues for an increase in space consumption. From our point of view, the notion of a universal plan is closely related to the notion of controllable languages developed for discrete event systems control [Ramadge and Wonham, 1989]. There, a language (a set of linear plans) is controllable iff the effects of all possible uncontrollable events do not lead outside the set of plans that this language contains. In other words, just as for Schoppers, all required system reactions to possible contingencies are compiled into the controllable language. Our framework is different, but complementary; it favors the on-line generation of appropriate reactions to exogenous events, as opposed to precompiling them into the Golog program.

ConGolog [De Giacomo *et al.*, 1997b, De Giacomo and Levesque, 1999b, De Giacomo *et al.*, 2000] is a much richer version of Golog that supports concurrency, prioritized interrupts and exogenous actions. Reactive behaviors are easily representable by ConGolog's interrupt mechanism, so that a combination of reactive behaviors with "deliberative" execution monitoring is possible. This would allow one to experiment with different mixtures of execution monitoring and reactivity, with the advantage of preserving the unifying formal framework of the situation calculus, but this remains an open research problem. The papers [Lespérance *et al.*, 1998, Lespérance and Ng, 2000] and [De Giacomo *et al.*, 2002] are important steps in this research direction. In particular, [Lespérance and Ng, 2000] considers a recovering mechanism based on backtracking in cases when all branches in a nondeterministic choice operator share the same initial sequence of robot actions. This has an advantage that if an exogenous action happens (e.g., the robot receives a new delivery request) when the robot has already executed some actions, then a new plan will be computed that will include the same sequence of already performed actions. This approach is based on execution semantics different from semantics considered in this chapter. Because in our case a postcondition characterizes what a Golog program has to achieve, new corrective actions can be inserted to undo effects of previously performed physical actions (if they are reversible) and/or parts of the program can be abandoned as long as the program postcondition will be satisfied when the program completes. But [Lespérance and Ng, 2000] does not use a postcondition to characterize a Golog program and considers actions in a Golog program as an "essential advice" that must be followed.

Several researchers have considered a combination of replanning and backtracking to repair a plan (e.g., see [Golden *et al.*, February 1996, Barruffi *et al.*, 1998, Stone and Veloso, 1999]). In particular, [Golden *et al.*, February 1996] introduces a term "backtracking over execution" as a reference to visiting a past plan in a search space after an action was physically executed. Our approach is different in several aspects. First, we propose a formal logical specification for monitored execution that has a recovery procedure based on combination of replanning and backtracking as a one possible implementation. Second, we consider backtracking not in the plan space, but backtracking to a past state of a nondeterministic Golog program. Third, we consider an example when actions have explicit temporal arguments; in addition, in our implementation, no "physical backtracking" (reversing effects of executed actions) is required because we consider a class of restartable Golog programs.

The theory of embedded planning [Traverso *et al.*, 1992, Giunchiglia *et al.*, 1994, Traverso and Spalazzi, 1 introduces notions of planning with failure and has motivations very similar to ours. The authors propose several formal languages that, like Golog, include constructs for sequence, conditionals, loops and recursion. The emphasis is on reactive programs, but their proposal does provide for replanning during execution.

Conformant planning [Goldman and Boddy, 1996, Cimatti and Roveri, 2000] can be described as the problem of finding a sequence of actions that is guaranteed to achieve the goal regardless of the nondeterminism of the domain. That is, for all possible initial states, and for all uncertain action effects, the execution of the plan must result in a goal state without gathering any run-time information. Our approach to dealing with uncertainty and incomplete information is different because we rely on run-time information during the execution of the Golog program and we do not try to compute in advance a sequence of actions that will achieve the goal no matter what exogenous actions will happen during the execution. As examples in this chapter have demonstrated, this allows us to compute on-line plans that lead to a goal state even when a conformant plan does not exist.

Several authors rely on formal theories of actions for the purposes of characterizing appropriate notions of action failures [Baral and Son, 1997, Sandewall, 1997], but they do not consider execution monitoring per se. [Bjäreland, 2001] considers a related (bit different) approach to execution monitoring when discrepancies can be attributed to incompleteness in the model, e.g., when the successor state axioms do not characterize all causal dependencies adequately and they have to be modified appropriately. He also describes several applications to real technical control systems. Control engineers often study the issues related to design of stable control systems, i.e., systems that will reach one of the desirable states within a finite number of transitions no matter what exogenous actions may happen and will continue to visit desirable states infinitely often after that. Using temporal logic, [Nakamura *et al.*, 2000] introduce and study a related but weaker concept of 'maintainability'.

Perhaps the most sophisticated existing plan execution monitor is the XFRM system of Beetz and McDermott [Beetz and McDermott, 1994, Beetz and McDermott, 1997]; XFRM is an extension of RPL (Reactive Plan Language). This provides for the continual modification of robot plans (programs) during their execution, using a rich collection of failure models and domain dependent plan repair strategies. The primary objective of our approach is to provide compact, declarative representations for the entire process of execution, monitoring and recovery from failure, and this chapter has presented our framework for this research program.

A domain-specific approach to monitoring, plan modification and rescheduling of autonomous robot plans is proposed in [Beetz and Bennewitz, 1998]. It is based on the idea of local planning of ongoing activities proposed in [Beetz and McDermott, 1996] and implemented in XFRM. Their notion of restartable plan and our notion of restartable program follow common intuitions: "we call a plan p restartable if for any task t that has p as its plan, starting the execution of p, causing its evaporation, and executing p completely afterwards causes a behavior that satisfies the task t" (see [Beetz and McDermott, 1996]). In contrast to our example, where the only exogenous action is sensing time, [Beetz and Bennewitz, 1998] considers sensing of doors (whether they are closed or not) in an office environment: plans are modified and rescheduled appropriately if exogenous actions of closing a door (or opening it) occur when the robot executes a plan. It remains to see what connections can be found between procedural approach advocated in [Beetz and McDermott, 1996, Beetz and Bennewitz, 1998] and our logical framework.

In the *CIRCA*[18] approach, the planner reasons about a fairly complex model of world dynamics and primitive control behaviors. Projecting the world dynamics forward, the planner synthesizes plans in a form called a Test-Action Pair schedule [Musliner *et al.*, 1995] and passes them to a real-time subsystem that executes plans reactively and enforces guaranteed response times. In our approach, instead of relying on plan synthesis methods, we obtain temporal plans as a result of interpreting a high-level program. An execution monitoring and recovery technique called *purely inserted recovery plans* is proposed in [Musliner *et al.*, 1991], a precursor of *CIRCA*. As we mentioned earlier in this chapter, it can be advantageous in the general case to combine purely inserted recovery plans with backtracking to a previous computation state to obtain a more flexible recovery method.

An execution monitoring architecture *RAMA*[19] proposed in [Earl and Firby, 1996] has a module *adapter* responsible for retrieving those programs from a library which are relevant to recovery from the expectation failure and for requesting a human assistance, if necessary. We take advantage of non-deterministic constructs in Golog and propose backtracking as another useful general recovery technique. The procedure "handle-expectation-failure" described there relies on a process combinator which is similar to $(\delta_1 \mid \delta_2)$ in Golog, but our backtracking technique may recover also in some cases when a Golog program contains another non-deterministic choice construct $\pi v.\delta$ unavailable in the process language of RAMA.

Dynamic programming approaches [Hansen and Cohen, 1992] to execution monitoring (called *recurrent deliberation* in [Dean *et al.*, 1995]) will give us an opportunity to improve monitoring strategies. In our case, a monitoring pace is determined by times when primitive actions are selected for execution, but between action occurrences the monitor does not attempt to estimate whether the current branch of a high-level program will complete before a deadline. In [Hansen and Cohen, 1992], monitoring tasks with deadlines is considered as a sequential decision problem, which makes available a dynamic programming method for constructing a decision rule for monitoring. However, the authors do not consider any recovery technique besides the decision to abandon a monitored task (if it is about to miss a deadline).

Rationale-based monitoring [Veloso *et al.*, 1998] and many other plan management issues [Horty and Pollack, 1998] are important research issues, but it remains to see how these issues may fit into our declarative approach towards execution monitoring of high-level programs.

In the area of requirements engineering, there are two complimentary approaches to man-

[18] *CIRCA* stands for Cooperative Intelligent Real-TIme Control Architecture.
[19] RAMA stands for Routine Activity Management and Analysis.

age runtime violations of requirements. First, anticipate as many as possible violations at specification time; more robust specifications can be derived from obstacles identified in the result of analysis of first-sketch specifications [Lamsweerde and Letier, 2000, Letier and van Lamsweerde, 2002]. Second, detect and resolve such violations at runtime, where resolution consists in making on-the-fly, acceptable changes to the requirements [Cohen *et al.*, May 1997, Feather *et al.*, April 1998]. In [Feather *et al.*, April 1998], the authors consider *reconciliation tactics* used in monitoring of breakable assertions and identify a few possible tactics: 1) enforce the assertion by introducing restorative actions on control parameters, 2) find an alternative assertion to achieve the same parent goal in the refinement graph, and 3) tune a parameter. The framework elaborated in this chapter is related more to the second approach, but it can be also interesting to make it useable along the lines of the first approach.

Finally, it is important to take into account how much time the interpreter and recovery mechanism take before they compute a new program that will be used to recover from a failure. The current framework assumes that the computation time is negligible. This is a very interesting direction for future research.

4.7 Conclusion

We provide logical specifications of on-line program executions (with monitoring) formulated in a version of the sequential situation calculus that includes time. Our account relies on specification of a single-step interpreter for the logic programming language Golog. This chapter makes several contributions: 1) it develops the situation calculus based framework for execution monitoring, 2) it extends our previously developed framework [De Giacomo *et al.*, 1998] by introducing trace and backtracking into the process of on-line monitoring of Golog programs, 3) it adapts the interpreter of [De Giacomo *et al.*, 1997b] to the temporal domain, 4) it develops a particular execution monitor in the case when the high level control module gets only temporal information as the sole sensory input from the external world. Thus, from the formal point of view, we develop a predicate logic framework for rescheduling and modification of temporal plans. From the robotics point of view, we develop a general architecture suitable for monitoring execution of restartable situation-calculus based programs written in Golog. A set of experiments on a real mobile robot Golem (B21 manufactured by RWI) demonstrated the usefulness of our architecture.

Chapter 5

Decision-Theoretic, High-level Programming.

In the preceding chapter, we considered the case when unmodelled uncertainty in an environment was accounted for by providing a mechanism that recovers from unanticipated contingencies intervening with an intended execution of a Golog program. The approach developed in that chapter is applicable even if a probabilistic model of exogenous actions (that can interfere with the agent actions) is not available and cannot be learnt on-line from interaction with an environment (e.g., if the agent has a very limited time to collect statistically valid experience). In this chapter, we would like to consider the case when a designer of a robot controller has both information about probability distributions of exogenous actions and information about rewards that characterize the quality of actions undertaken by a controller. The availability of a rich information model allows us to exploit well known techniques of decision-theoretic planning under an assumption that an interaction of a robot controller with a stochastic environment can be characterized in terms of fully observable Markov Decision Processes (MDP). In this chapter, we consider a framework for robot programming which allows the seamless integration of explicit agent programming with decision-theoretic planning. Specifically, we consider a novel decision-theoretic Golog (*DTGolog*) model that allows one to partially specify a control program in Golog, and provides an interpreter that, given a situation calculus axiomatization of a domain, will determine the optimal completion of that program (viewed as a Markov decision process). We demonstrate the utility of this model with results obtained in an office delivery robotics domain.

5.1 Introduction

The construction of autonomous agents, such as mobile robots or software agents, is paramount in artificial intelligence, with considerable research devoted to methods that will ease the burden of designing controllers for such agents. There are two main ways in which the conceptual complexity of devising controllers can be managed. The traditional approach to the design of controllers solving decision-theoretic planning task relies on a state-based representation. The task of a controller is to compute an optimal policy that specifies what primitive action must be executed in every state. However, this framework has a number of well-known difficulties associated with the size of a state space that grows exponentially with the number of state features and with the complexity of computing an optimal primitive action for every state. An alternative approach to designing controllers that must operate efficiently in stochastic environments is to provide languages with which a programmer can specify a control program with relative ease, using high-level actions as primitives, and expressing the necessary operations in a natural way. Unfortunately, this second approach can be conceptually unmanageable for human programmers who are responsible for solving a non-trivial decision-theoretic task and implementing the solution. In this chapter, we consider a framework that combines both perspectives, allowing one to partially specify a controller by writing a program in Golog, yet allowing an agent some latitude in choosing its actions, thus requiring a modicum of planning or decision-making ability. Viewed differently, we allow for the seamless integration of programming and planning. Specifically, we suppose that the agent programmer has enough knowledge of a given domain to be able to specify some (but not necessarily all) of the structure and the details of a good (or possibly optimal) controller. Those aspects left unspecified will be filled in by the agent itself, but must satisfy any constraints imposed by the program (or partially-specified controller).

There are two main reasons why Golog is a well suited language for specifying efficient decision-theoretic controllers. First, primitive actions in Golog are axiomatized in terms of basic action theories which represent an application domain in terms of individual properties rather than a collection of states. For this reason, the situation calculus based representation of an application domain is more economical: it grows linearly with the number of new properties that need to be represented. In addition, because the situation calculus is a predicate logic language, domains can be represented succinctly by relying on quantified expressions instead of the propositional representations that are commonly used to deal with factored MDPs (such as algebraic decision diagrams, dynamic Bayesian networks, PSTRIPS mentioned in the end of

Section 2.4.1 and extensively discussed in [Boutilier *et al.*, 1999]). Second, Golog has several nondeterministic choice operators, and a Golog program can be viewed as a partially-specified controller. The agent (or interpreter) is free to choose any valid execution of the program consistent with the constraints it imposes. This provides an opportunity to design a Golog interpreter that considers all non-deterministic choices in a program as alternatives similar to branches in a decision tree (see Section 2.4.2) and makes a choice of a branch that corresponds to the highest expected accumulated reward. However, if a Golog program uses an operator that specifies a unique next action to be executed, then significant computational efforts can be saved because a controller does not need to deliberate what next action is an optimal action to do.

In this chapter, we consider DTGolog, which extends Golog (reviewed in Section 2.3) in two ways. First, we allow actions to be stochastic. Second, we refine the interpreter so that nondeterministic choices are made using decision-theoretic optimization criteria. The interpreter makes choices that maximize expected utility *given* that the interpreter is committed to executing the program provided. In this way, we add decision-theoretic planning capability to the Golog interpreter, allowing it to optimally "complete" the partially-specified controller. The agent can only adopt policies that are consistent with the execution of the program. The decision-theoretic Golog interpreter then solves the underlying MDP by making choices regarding the execution of the program through expected utility maximization. As such, the interpreter is solving a *constrained MDP*: it produces a policy that is optimal subject to the constraint that the policy implements the given program. The algorithm used by the DTGolog interpreter is based on standard, forward dynamic programming (or decision tree) models for solving MDPs.

The field of robotics and autonomous agents, toward which this research is ultimately directed, has been torn between two major programming paradigms: the model-based planning approach [Arkin, 1989, Nilsson, 1969, Fikes *et al.*, 1972] and the *behavior-based* or *reactive* paradigm [Brooks, 1989]. Both approaches have their advantages (flexibility and generality in the planning paradigm, performance of reactive controllers) and limitations (e.g., the requirement for domain models and the intrinsic complexity of planning approaches, task-specific design and conceptual complexity for programmers in the reactive model). Contrasting both methods — control through planning and control through programming controllers — suggests that *combining both* should be the preferred way to go: when controllers can easily be designed by hand, planning has no role to play. On the other hand, certain robotics problems are much easily treated through the specification of a domain theory. The seamless integration of programming and planning, thus, has great promise for the field of robotics.

We describe the DTGolog representation of MDPs and programs and the DTGolog interpreter in the subsequent sections, and illustrate the functioning of the interpreter by describing its implementation in a office robot in the last section. The Prolog code of the DTGolog interpreter is provided in Appendix C.

5.2 DTGolog: Decision-Theoretic Golog

We now turn our attention to DTGolog, a decision-theoretic extension of the Golog framework that allows one to specify MDPs in a first-order language, and provide "advice" in the form of high-level programs that constrain the search for good or optimal policies.

As a planning model, MDPs are quite flexible and robust, dealing with uncertainty, multiple objectives, and so on, but suffer from several key limitations. First, little work has gone into the development of first-order languages for specifying MDPs (see [Bacchus *et al.*, 1995, Poole, 1997, Boutilier *et al.*, 2001] for exceptions). Second, the computational complexity of optimal policy construction is prohibitive [Littman *et al.*, 1995, Papadimitriou and Tsitsiklis, 1987, Littman *et al.*, 1998, Mundhenk *et al.*, 2000]. As mentioned, one way to circumvent planning complexity is to allow explicit agent programming; yet little work has been directed toward integrating the ability to write programs or otherwise constrain the space of policies that are searched during planning. What work has been done (e.g., [Parr and Russell, 1998, Sutton, 1995]) fails to provide a language for imposing such constraints, and certainly offers no tools for *programming* agent behavior. We believe that natural, declarative *programming languages and methodologies* for (partially) specifying agent behavior are necessary for this approach to find successful application in real domains. More detailed comparison with the previous work is provided at the end of this chapter.

5.2.1 DTGolog: Problem Representation

The specification of an MDP requires the provision of a Reiter-style *basic action theory*—as in Section 2.1.2—and a background *optimization theory*—consisting of the specification of a reward function and some optimality criterion (here we require only a finite horizon H). We choose the temporal situation calculus to provide a basic action theory because it has a richer ontology and more expressive language than the basic situation calculus in Section 2.1.1 and this additional expressivity proves to be useful in specifying Golog controllers for mobile robots. The unique names axioms and initial database have the same form as in standard Golog. To avoid extending the logical language of the temporal situation calculus, we introduce MDP-related concepts using definitions (we call them also abbreviations). This is important because we would like all previously proved theorems about the situation calculus continue to apply to those basic action theories which formalize an MDP. It is a common practice to construct a logical theory T_1 from another logical theory T_0 by adding a new defined function symbol or a new defined predicate symbol [Enderton, 2001]. The representation considered in this section is originally proposed in our paper [Boutilier *et al.*, 2000a], but below we use also representational tricks formulated in [Reiter, 2001a].

A basic action theory for MDPs.

As we know, in Reiter's basic action theories we have successor state axioms and precondition axioms for deterministic actions only. Because we would like to specify MDPs that include stochastic actions, it seems at a first glance that we have to extend the ontology of the situation calculus and introduce a new sort for stochastic actions. However, if we want to rely on the usual basic action theories, then we can imagine that whenever an agent initiates a stochastic action A in s, nature steps in and chooses nondeterministically one of the possible outcomes N_i: each of these outcomes is a primitive deterministic action and the situation $do(N_i, s)$ is considered as one of resulting situations. Nature's actions cannot be executed by the agent itself and the agent has no control over which deterministic action nature will do. An outcome of a stochastic action is not known in advance simply because we do not know which particular deterministic action nature will choose, but still we can consider outcomes as deterministic actions that are enabled by the agent. Formally speaking, we follow the representational trick suggested in [Reiter, 2001a]: stochastic actions are introduced using a new defined symbol $choice(A, n, s)$, where A is a stochastic action, n is one of nature's actions in situation s. We do this because there is no reason to introduce any modifications (such as stochastic actions) in the ontology of an underlying temporal basic action theory: it has sorts for situations, deterministic primitive actions, time and objects, and this ontology is sufficient for our needs. As a consequence, when we consider nature's actions we can use standard precondition axioms and standard successor state axioms that mention only deterministic agent actions and nature's actions (which are deterministic). In other words, precondition and successor state axioms never mention stochastic agent actions; but stochastic agent actions can occur in Golog programs and in policies (which are Golog programs of a special form that will be defined later). Thus, we would like to inherit as much as possible the usual approach to the axiomatization of basic action theories and the usual structure of Golog interpreters, where we always check preconditions of an action before we determine the transition to the next situation from the current situation. Note also that this macro approach to stochastic actions is similar to the approach for complex actions: we never need precondition axioms for procedures, if-then-else, nondeterministic and other complex actions.

Let us consider the case when all stochastic actions have only a finite number of outcomes and assume that the stochastic action A has m different available outcomes $N_1, \ldots, N_m$ if a certain logical condition $\psi(s)$ holds in s. In general, the number and type of outcomes can vary from situation to situation. Let $\phi_1(s), \ldots, \phi_k(s)$ be the set of mutually disjoint logical conditions which are situation calculus formulas uniform in s and such that $\phi_1(s) \vee \ldots \vee \phi_k(s)$ is true for any s. Then, the domain theory includes

$$\begin{aligned} choice(A, a, s) \stackrel{def}{=} \phi_1(s) \supset (a = N_1^1 \vee \cdots \vee a = N_m^1) \wedge \\ \cdots \wedge \\ \phi_k(s) \supset (a = N_1^k \vee \cdots \vee a = N_m^k). \end{aligned} \tag{5.1}$$

In the sequel, we will consider only the simplest case when every stochastic action has always the same outcomes available in every situation[1]:

$$choice'(A) \stackrel{def}{=} \{N_1, \ldots, N_m\}.$$

Moreover, when a stochastic action A is executed by the agent, nature chooses one of the associated actions N_i with a specified probability, and the resulting situation is determined by nature's action so chosen. Let *prob*(n, a, s) denote the probability with which nature chooses n in s when the agent does stochastic action a. In the simplest case, probabilities of nature's choices remain the same in any situation, i.e., for each stochastic action A, the domain axiomatization includes the following:

$$prob(N_1, A, s) = p_1 \stackrel{def}{=} p_1 = P_1 \ , \ \ldots \ , \ prob(N_m, A, s) = p_m \stackrel{def}{=} p_m = P_m,$$

where $P_1, \ldots, P_m$ are probabilities that sum to 1. In a more general case, right-hand sides of definitions may include arbitrary formulas uniform in s. It seems reasonable to require a certain correspondence between probability of an outcome n and logical preconditions for occurrence of this outcome in situation s. More specifically, if n is not possible in s, then the probability

[1] In cases, when there are too many independent outcomes, e.g., the $spray10parts$ stochastic action can spray paint each of 10 different parts independently with the probability 0.9, one can introduce concurrent actions to avoid enumerating a large number of nature's actions [Reiter, 2001a]. One can use also representation transformation techniques considered in [Dearden and Boutilier, 1997, Littman, 1997].

of this outcome must be 0 and vice versa. In addition, for a stochastic action A, if none of its outcomes is physically possible in s, then probabilities of outcomes are defined in s as 0. In a general case, following [Reiter, 2001a], we require that the probability distributions must be defined properly when the axiomatizer formalizes a probabilistic domain. One needs to verify the following two properties: for any stochastic action A and its associated nature's choices $N_1, \ldots, N_m$

$$(\mathit{Poss}(N_1, s) \vee \cdots \vee \mathit{Poss}(N_m, s)) \supset \mathit{Poss}(N_i, s) \equiv prob(N_i, A, s) > 0, \;\; i = \{1, \ldots, m\} \quad (5.2)$$

$$(\mathit{Poss}(N_1, s) \vee \cdots \vee \mathit{Poss}(N_m, s)) \supset \sum_{i=1}^{m} prob(N_i, A, s) = 1. \quad (5.3)$$

According to (5.2) and (5.3), if any of nature's actions is possible, then the sum of the probabilities of all possible actions must be 1. Otherwise, if none of nature's actions N_i is physically possible in s after an agent executes a stochastic action A, then we do not require that the sum is equal to 1: all values $prob(N_i, A, s)$ are defined in this situation s as 0. For example, if an agent holds a coin then after tossing this coin nature can choose one of the two outcomes "heads up" or "tails up" and probabilities of these outcomes sum to 1, but if both nature's actions are not possible (e.g., the agent does not hold the coin anymore because the agent put the coin on a table and moved to another room), then we no longer require that probabilities must sum to 1.[2] We provide detailed examples of axiomatizations of probabilistic domains later in this chapter. Note that deterministic agent actions can be considered as stochastic agent actions with only one choice available for nature. However, for implementational reasons, it is convenient to treat them separately.

The background action theory also includes a new class of definitions, *sense conditions* definitions, which assert situation suppressed formulae using an abbreviation *senseCond*(n, ϕ): this holds if ϕ is a situation-suppressed (see Definition 2.2.7) logical expression that an agent uses to determine if the specific nature's action n occurred when some stochastic action was executed. We require such definitions in order to "implement" full observability. While in the standard fully observable MDP model one simply assumes that the successor state is known, in practice, one must force agents to disambiguate the state using sensor information. Indeed, when the agent does a stochastic action A in situation s the agent does not know what action n nature will choose in s. Because we assume that each stochastic action A has the same outcomes in all situations, it might seem that it should be sufficient for

[2]This approach of assigning probabilities (with the sum equal to 1) to nature's actions only in the case when at least one nature's action is physically possible seems to be in concordance with notions of randomness and probability discussed in [Kolmogorov, 1983, Fine, 1973, Li and Vitanyi, 1997].

the agent to evaluate mutually disjoint conditions $\phi_1(do(n, s)), \ldots, \phi_m(do(n, s))$ mentioned in *senseCond*$(N_1, \phi_1), \ldots,$ *senseCond*(N_m, ϕ_m) to find out nature's action in $do(n, s)$. It is guaranteed that exactly one of them evaluates to true in any situation s:

$$(\forall i, j \in \{1, \ldots, m\}) \;\; i \neq j \supset \neg(\phi_i(s) \wedge \phi_j(s)) \; , \;\; \phi_1(s) \vee \ldots \vee \phi_m(s) \tag{5.4}$$

But these evaluations cannot be directly implemented with respect to $do(n, s)$ because the agent does not know the value of n. However, if the agent does a sequence of sensing actions sufficient to determine what unique condition ϕ_i holds in the situation resulting from doing these sensing actions, then the agent learns that $n = N_i$. The sensing actions needed for each stochastic action A can be determined from axioms that define the procedure *senseEffect*(A). These domain specific axioms define procedures that consist of one or a sequence of prespecified sensing actions that will be executed after a stochastic action A, where sensing actions are action terms as considered in Chapter 3. The domain axiomatizer is responsible for providing these *senseEffect* axioms defining which sensing action(s) the agent has to execute to gather required information from sensors.[3] Note that the form of sense conditions definitions, *senseCond*(N_i, ϕ_i) indicates that we assume that the same condition ϕ_i needs to be evaluated after doing sensing actions specified by *senseEffect*. In other words, even if a given N_i can have different effects depending on the situation in which it occurs, we assume in the sequel that the specified sensing actions will provide information sufficient to identify uniquely the outcome of the last stochastic action by evaluating expressions $\phi_1(s), \ldots, \phi_m(s)$ in the situation s that results from sensing.

Finally, we would like to note that the standard MDPs have no concept of time associated with transitions from one state to another. In the temporal situation calculus that we use as the background action theory in this chapter, actions are instantaneous and they can either initiate or terminate processes extended over time (these processes are represented by fluents). Lower bounds on durations of these processes can be specified in the domain axiomatization and exact values of durations can be determined by a Golog interpreter from constraints provided together by the domain axiomatization and by a Golog program as we discussed in Sections 2.1.2 and 2.3.1. Note that from the MDP perspective, moments of time when instantaneous actions are executed can be considered as yet another factor (state variable) in addition to other state variables (represented by fluents). They should not be confused with moments of time mentioned often in MDP literature where they are understood as stages or decision epochs. In

[3] If it is necessary, these *senseEffect* axioms can be generated automatically.

the temporal situation calculus, decision epochs correspond to situation terms.[4]

A background optimization theory.

A decision-theoretic optimization theory contains axioms specifying the reward function.[5] In their simplest form, reward axioms use a new defined function symbol $reward(s)$ and assert costs and rewards as a function of the action taken, properties of the current situation, or both (note that the action taken can be recovered from the situation term). For instance, we might assert

$$reward(do(giveCoffeeS(Jill,t),s)) \stackrel{def}{=} 6.3$$

Because primitive actions have an explicit temporal argument, we can also describe time-dependent reward functions easily. This often proves useful in practice. Recall that in a given temporal Golog program, the temporal occurrences of some actions can be uniquely determined either by temporal constraints or "hard-coded" by the programmer. Other actions may occur at any time in a certain interval determined by temporal inequalities; for any such action $A(\vec{x},t)$, we can instantiate the time argument by a value that maximizes the reward for reaching the situation $do(A(\vec{x},t),s)$. For example, suppose the robot receives a small reward 1 in the situation $do(startGo(l_1,l_2,t_i),s_i)$ and a reward $\max\limits_s(\frac{100-t}{distance(l_1,l_2)})$ for doing the action $endGoS(l_1,l_2,t)$ in s, where the maximization of the shown linear function of t is taken with respect to all temporal inequalities constraining time occurrences of actions mentioned in the situation term s. (See Section 4.4.4 for a more detailed discussion of a similar example.) With this reward function, the robot is encouraged to arrive at the destination as soon as possible (the smaller is the value of time t when *endGoS* is executed, the higher is the reward) and is also encouraged to go to nearby locations (because the reward is inversely proportional to distance). Although sensing actions can incur significant costs in a general case, in the examples considered later in this chapter, the robot receives rewards only for doing physical actions. We will see later, that taking into account costs of sensing actions can be nontrivial in the current version of the DTGolog interpreter, because the agent might wish to avoid sensing to get a higher accumulated expected reward, but this may lead to the partially observable case that we

[4] In a more general setting, durations of processes can be modeled as random variables and behavior of a decision-making agent should be modeled by a well known extension of the MDP model – semi-Markov Decision Processes (SMDP) – that allows one to account for a random time taken by a transition processes [Puterman, 1994, Çinlar, 1975, Korolyuk and Korolyuk, 1999, Parr, 1998b]. However, in the sequel, to simplify our presentation we always assume that time taken by transitions between consecutive states does not depend on random factors and the structure of the overall decision process – a sequence of instantaneous decisions leading to transitions to one of the successor states – can be approximated as a simple MDP.

[5] We require an optimality criterion to be specified as well. We assume a finite-horizon H in this chapter.

do not consider in this thesis.

Given the collection of new macros and axioms introduced in this section we can now define policies precisely. Our inductive definition will be a predicate logic version of the definition of policy introduced in Section 2.4. This definition uses auxiliary deterministic action constants *Stop* and *Nil*: both actions are possible to execute in any situation, they have no effects on any fluent, both actions take the agent into an absorbing state, and the agent incurs zero cost for doing both actions in any situation and receives a zero reward for being in an absorbing state. Intuitively, *Stop* means that execution must be abnormally terminated by an agent[6] and *Nil* means "do nothing": the agent can repeat this action and stay idle until horizon reaches 0 (if ever).

Definition 5.2.1: A policy is a Golog program inductively defined by:

1. Any agent deterministic action or auxiliary action *Stop* or *Nil* is a policy.[7]

2. If A is an agent deterministic action and π is a policy, then $(A;\ \pi)$ is a policy.

3. If α is an agent stochastic action and $N_1, \ldots, N_k$ is a subset of nature's choices associated with α (if it is an empty subset, then $k = 0$), *senseEffect*(α) is a procedure specifying sensing actions that the agent does to observe its environment, *senseCond*(N_1, ϕ_1), ..., *senseCond*(N_k, ϕ_k) are expressions specifying situation suppressed mutually disjoint formulas $\phi_1, \ldots, \phi_k$, and $\pi_1, \ldots, \pi_k$ are policies, then each of the following expressions is a policy:

$$
\begin{array}{lll}
(k=0) & \alpha \ ; \ \textit{senseEffect}(\alpha) \ ; & Stop \\
(k=1) & \alpha \ ; \ \textit{senseEffect}(\alpha) \ ; & (\phi_1)? \ ; \ \pi_1 \\
(k=2) & \alpha \ ; \ \textit{senseEffect}(\alpha) \ ; & \textbf{if } \phi_1 \textbf{ then } \pi_1 \textbf{ else } (\phi_2)? \ ; \ \pi_2 \\
(k\geq 3) & \alpha \ ; \ \textit{senseEffect}(\alpha) \ ; & \textbf{if } \phi_1 \textbf{ then } \pi_1 \\
 & & \quad \vdots \\
 & & \textbf{else if } \phi_{k-1} \textbf{ then } \pi_{k-1} \\
 & & \textbf{else } (\phi_k)? \ ; \ \pi_k
\end{array}
$$

Note that our definition deviates from the usual concept of a (nonstationary) Markov policy; in the MDP literature, a Markov policy is a function mapping each state (and a decision epoch)

[6] This can be viewed as having an agent simply give up its attempt to execute the policy and await further instruction.

[7] Because a policy is a Golog program for the agent who cannot execute nature's actions, only deterministic agent actions can occur in a policy.

to an action. Because we are interested only in policies that the agent can start executing in a certain initial state (that corresponds to S_0), our policies are conditional Golog programs that prescribe an action only in states reachable from the initial state. Nevertheless, Definition 5.2.1 seems a natural extension of the conventional definition to a case when a decision maker is interested only in a certain relevant subset of the state space.

We would like to define also branches of a policy and leaves of branches.

Definition 5.2.2: Let π be a policy. A *branch* of π is defined inductively as follows.

1. If π is a deterministic agent action or *Nil*, or *Stop*, then the only branch of π is this action (or auxiliary constant).

2. Let A be a deterministic agent action. The sequential composition $(A\ ;\ B)$ is a branch of π iff π is a policy $(A\ ;\ \pi')$, and B is a branch of π'.

3. Let α be a stochastic agent action, $N_1, \ldots, N_k$ be a subset of the set of nature's choices associated with α, *senseEffect*(α) be a sequence of sensing actions, *senseCond*(N_1, ϕ_1), ..., *senseCond*(N_k, ϕ_k) be sense conditions, and $\pi_1, \ldots, \pi_k$ be policies. If π is a policy $(\alpha\ ;\ \textit{senseEffect}(\alpha)\ ;\ Stop)$, then this sequence is the only branch of π. If π is a policy $(\alpha\ ;\ \textit{senseEffect}(\alpha)\ ;\ (\phi_1)?\ ;\ \pi_1)$ and B_1 is a branch of π_1, then the sequence $(\alpha\ ;\ \textit{senseEffect}(\alpha)\ ;\ (\phi_1)?\ ;\ B_1)$ is a branch of π. If π is a policy
$$(\alpha\,;\ \textit{senseEffect}(\alpha)\,;\ \textbf{if}\ \ \phi_1\ \textbf{then}\ \ \pi_1 \cdots \textbf{else}\ \ \textbf{if}\ \ \phi_{k-1}\ \textbf{then}\ \ \pi_{k-1}\ \textbf{else}\ \ (\phi_k)?\ ;\ \pi_k)$$
and $B_1, \ldots, B_k$ are branches of, respectively, $\pi_1, \ldots, \pi_k$, then each of the sequences $(\alpha\,;\ \textit{senseEffect}(\alpha)\,;\ (\phi_1)?\,;\ B_1)$, ..., $(\alpha\,;\ \textit{senseEffect}(\alpha)\,;\ (\phi_k)?\,;\ B_k)$ is a branch of π.

4. Thus, branches of a policy π are simply sequences of actions and test expressions. The *leaf* of a branch B is the last action that occurs in B. Note that for any branch B, the leaf of B is either a deterministic agent action, or *Nil*, or *Stop*.

Finally, we define a value of an initial state with respect to a branch of a policy and define also a probability associated with a branch that is executed starting in an initial state. In the sequel, we call the probability associated with a branch simply the probability of a branch (with respect to an initial state). We skip the phrase 'with respect to a state X', whenever it is clear from the context what state is being considered. The probability of a branch will be used later to characterize policies in terms of their 'executability'. Later, we define also a value of an initial state with respect to a policy. This notion parallels a well known definition of value function for an initial state with respect to a given policy. In the following definition, we assume that we are given a basic action theory and optimization theory $\mathcal{D}$ that provide axiomatization of an MDP; truth of formulas is understood as entailment from the conjunction of these two theories.

Definition 5.2.3: Let B be a branch of a policy and S be any ground reachable situation (see the definition (2.9) and also Corollary 2.2.6). In addition, let $prob(a, n, s)$ be probabilities of nature's choices n for each stochastic agent action a and $reward(do(a, s))$ be a reward function for doing an action a in s. The *probability* and *value* of B with respect to S are defined inductively as follows.

1. Let B be a deterministic agent action A. If A is possible in S, then the probability of this branch is 1 and the value with respect to S is $reward(S) + reward(do(A, S))$. If A is not possible in S, then the probability of this branch is 0 and the value is $reward(S)$. If B is *Stop* (or *Nil*), then the probability of this branch is 0 (1, respectively) and the value of this branch is $reward(S)$, in both cases.

2. Let B be a branch $(A\ ;\ B')$, where A is a deterministic agent action and B' is a branch. If $Poss(A, S)$ is true with respect to a background theory, the action A is possible in S, then the probability of B with respect to S is equal to the probability of the branch B' with respect to $do(A, S)$ and the value of B with respect to S equals the sum of $reward(S)$ and the value of B' with respect to $do(A, S)$. If the action A is not possible in S, then the probability of B is 0 and the value of B is equal to $reward(S)$.

3. Let B be a branch $(\alpha\,;\ \mathit{senseEffect}(\alpha)\,;\ Stop)$, where α is a stochastic agent action, then the probability of this branch is 0 and the value of this branch with respect to S is $reward(S)$.

4. Let B be a branch $(\alpha\,;\ \mathit{senseEffect}(\alpha)\,;\ \phi?;\ B')$, where B' is a branch, α is a stochastic agent action, $\mathit{senseEffect}(\alpha)$ is a sequence of primitive sensing actions $A_1; \ldots; A_l$, ϕ is a test expression that occurs in $\mathit{senseCond}(N, \phi)$ such that N is one of choices available for nature when the agent executes α. Recall that $do([A_1; \ldots; A_l], S)$ is the compact notation introduced in Section 2.1.1. If $\phi(do([A_1; \ldots; A_l], do(N, S)))$ is true, then the probability of B with respect to S is equal to the product $prob(\alpha, N, S) \cdot p'$, and the value of B with respect to S equals the sum $reward(S) + prob(\alpha, N, S) \cdot r'$, where p' is the probability of B' with respect to $do([A_1; \ldots; A_l], do(N, S))$ and r' is the value of the branch B' with respect to $do([A_1; \ldots; A_l], do(N, S))$. If the formula $\phi(do([A_1; \ldots; A_l], do(N, S)))$ is not true, then the probability of B is 0 and the value of B is equal to $reward(S)$. Note that the value is adjusted by probability of nature's action.

Definition 5.2.4: Let π be a policy and S be a reachable ground situation. The success probability $P_S(\pi)$ of the policy π (with respect to S) is the total probability (with respect to S) of all branches in π that have leaves different from *Stop*.

Note that each branch with *Stop* as the leaf has the probability 0; nevertheless, we mention those branches explicitly in the definition to emphasize that they are associated with 'abnormal' termination leading to the absorbing state. It is easy to observe that according to the above definitions of the probability of a branch with respect to S and abbreviation $prob(a, n, s)$, we have that $0 \leq P_S(\pi) \leq 1$ for any policy π. In the sequel, we call also $P_S(\pi)$ the probability of successful execution of π.

Definition 5.2.5: Let π be a policy and S be a reachable ground situation. The value V_S^π of the policy π (with respect to S) is the total value (with respect to S) of all branches in π.

As usual in MDPs, the decision problem faced by an agent is that of computing an *optimal policy* that maximizes expected total accumulated reward. In the sequel, we consider only finite horizon MDPs mentioned in Section 2.4.

Example 5.2.6: Let us consider a simple example to illustrate the representation introduced in this section (we ignore temporal arguments in action terms to simplify the example). There is one stochastic action $flip(x)$ that flips a coin x:

$$choice'(flip(x)) \stackrel{def}{=} \{flipHead(x), flipTail(x)\}.$$

Nature's actions are always possible: $Poss(flipHead(x), s)$ and $Poss(flipTail(x), s)$. Their probabilities are $prob(flipHead(x), flip(x), s) \stackrel{def}{=} 0.5$ and $prob(flipTail(x), flip(x), s) \stackrel{def}{=} 0.5$. We assume that the procedure $senseEffect(flip(x))$ is defined as

$$\textbf{proc}\ senseEffect(flip(x))\ \ (\pi v).sense(head(x), v)\ \textbf{endProc}$$

To find out the outcome of the stochastic action $flip(x)$ the agent senses the position of the coin x: if it is heads up, then the agent's sensor returns the binary value YES, but if the agent observes that the coin x is tails up, then the sensor returns the binary value NO. We assume that both sense outcomes are possible in any situation, but the actual result of sensing will be determined only at the run time, when the agent executes the policy.

There is one fluent: the predicate $head(c, s)$ is true if the coin c is heads up in s. In S_0, we have $\neg head(x, S_0)$ for all coins x. The successor state axiom is the following:

$$\begin{array}{c} head(coin, do(a, s)) \equiv a = flipHead(coin) \lor a = sense(head(x), \text{YES}) \lor \\ head(coin, s) \land a \neq flipTail(coin) \land a \neq sense(head(x), \text{NO}). \end{array}$$

Sense conditions are defined by the following expressions: *senseCond*$(flipHead(x),\ head(x)\ )$ and *senseCond*$(flipTail(x),\ (\neg head(x))\ )$. Let us assume that the agent has five coins and the state space in this example are vertices of the 5-dimensional boolean cube, where 1 corresponds to the case when a coin is heads up. The initial state is the node $(0, 0, 0, 0, 0)$. Because the agent can flip only one coin at a time, the transitions in this state space happen only between adjacent vertices or from a node to itself. To complete the definition of the MDP, we have to define a reward function as a function of a previous state and the most recent action.[8] In any state, if flipping a coin yields tail, then the reward is 0. In any state where at least one coin is heads up, if the agent flips the first coin and gets head, then the reward is -10, but in the state $(0, 0, 0, 0, 0)$ if the agent gets head after flipping the first coin, then the reward is $+100$. In the state $(1, 0, 0, 0, 0)$, if flipping the third coin yields heads, then the reward is $+300$; in all other states, this outcome has the reward -30. In the state $(1, 0, 1, 0, 0)$, if flipping the fifth coin yields heads, then the reward is $+500$; in all other states, this outcome has the reward -50. In the state $(1, 0, 1, 0, 1)$, flipping the head of the second coin brings the reward 200, in the state $(1, 1, 1, 0, 1)$, flipping the head of the fourth coin yields the reward 400, but in all other states different from $(1, 0, 1, 0, 1)$, or different from $(1, 1, 1, 0, 1)$, respectively, flipping the head of the second coin (of the fourth coin, respectively), yields the reward -20 (-40, respectively).

[8] We are slightly abusing terminology and talk about rewards (instead of costs) even when they are negative.

This reward function is defined by the following axiom:

$$
\begin{array}{l}
\textit{reward}(do(a,s)) = v \stackrel{def}{=} (\forall x) a = flipTail(x) \wedge v = 0 \vee \\
\quad a = flipHead(C1) \wedge \\
\qquad ((\exists x) head(x,s) \wedge v = -10 \vee \neg(\exists x) head(x,s) \wedge v = 100) \vee \\
\quad a = flipHead(C3) \wedge \\
\qquad (head(C1,s) \wedge \neg head(C2,s) \wedge \neg head(C3,s) \wedge \neg head(C4,s) \wedge \neg head(C5,s) \wedge v = 300 \vee \\
\qquad\quad (\neg head(C1,s) \vee head(C2,s) \vee head(C3,s) \vee head(C4,s) \vee head(C5,s)) \wedge v = -30) \vee \\
\quad a = flipHead(C5) \wedge \\
\qquad (head(C1,s) \wedge head(C3,s) \wedge \neg head(C2,s) \wedge \neg head(C4,s) \wedge \neg head(C5,s) \wedge v = 500 \vee \\
\qquad\quad (\neg head(C1,s) \vee \neg head(C3,s) \vee head(C2,s) \vee head(C4,s) \vee head(C5,s)) \wedge v = -50) \vee \\
\quad a = flipHead(C2) \wedge \\
\qquad (head(C1,s) \wedge head(C3,s) \wedge head(C5,s) \wedge \neg head(C2,s) \wedge \neg head(C4,s) \wedge v = 200 \vee \\
\qquad\quad (\neg head(C1,s) \vee \neg head(C3,s) \vee \neg head(C5,s) \vee head(C2,s) \vee head(C4,s)) \wedge v = -20) \vee \\
\quad a = flipHead(C4) \wedge \\
\qquad (head(C1,s) \wedge head(C3,s) \wedge head(C5,s) \wedge head(C2,s) \wedge \neg head(C4,s) \wedge v = 400 \vee \\
\qquad\quad (\neg head(C1,s) \vee \neg head(C3,s) \vee \neg head(C5,s) \vee \neg head(C2,s) \vee head(C4,s)) \wedge v = -40)
\end{array}
$$

In the initial situation, $reward(S_0) \stackrel{def}{=} 0$. A Prolog implementation of this coin example and the optimal policy for horizon 6 are provided in Section C.2 and Section C.3 of Appendix C.

5.2.2 DTGolog: Semantics

In what follows, we assume that we have been provided with a background action theory and optimization theory. We interpret DTGolog programs relative to this theory. DTGolog programs are written using the same program operators as Golog programs. The semantics is specified in a similar fashion, with the predicate *BestDo* (axiomatized below) playing the role of *Do*. Specifically, the semantics of a DTGolog program is defined by a predicate *BestDo*(p, s, hor, π, v, pr), where p is a Golog program, s is a starting situation, π is the optimal policy determined by program $prog$ beginning in situation s, v is the total accumulated expected value of that policy, pr is the probability that π will execute successfully, and hor is a prespecified horizon (the expected values are computed in the usual way by multiplying rewards by probabilities of outcomes). Generally, an interpreter implementing this definition will be called with a given program $prog$, situation S_0, and a finite horizon hor, and the arguments π, v and pr will be instantiated by the interpreter. In the sequel, we simply call π the optimal policy whenever it is clear from the context that π is a policy optimal with respect to a given program and a finite horizon.

However, the structure of *BestDo* (and its Prolog implementation) is rather different than that of *Do*. One difference reflects the fact that primitive actions can be stochastic. The interpreter computes not just a ground situation term, but a policy. Furthermore, when the stochastic primitive action dictated by a program is considered by the *BestDo* interpreter, one action outcome may preclude continuation of the program, while another may lead to successful completion. If an agent enters a situation where the next step of its program is impossible (because either a test expression evaluates to false or a precondition of the next action is false), we indicate this by inserting into the policy a zero-cost *Stop* action that takes it to an absorbing state.

A second difference has to do with the optimization theory. In the original Golog framework (see Section 2.3.1), no criteria are used to distinguish one ground situation term computed by the interpreter from another. In particular, nondeterministic branches chosen by the interpreter in a program are not distinguished as long as they allow the interpreter to compute a ground situation term. In our decision-theoretic setting, we want nondeterministic choices to be made in a way that maximizes expected accumulated reward. Given a choice between two actions (or subprograms) at some point in a program, the interpreter chooses the branch with highest expected value, mirroring the structure of an MDP search tree. Intuitively, then, the semantics of a DTGolog program will be given by the *optimal execution of that program*, where optimality is defined in much the same way as for MDPs. Viewing the program as advice — or a constraint on the set of policies we are allowed to consider — the DTGolog interpreter provides us with an optimal policy under the constraints specified by the program.

Before discussing the details of the semantics, we note that there is some conflict between the Golog perspective and the MDP perspective in interpreting a DTGolog program. We can illustrate this with a very simple example.

First, we start with the MDP perspective. Suppose program $\delta = (\delta_1|\delta_2)$, where the subprograms δ_1 and δ_2 each contain stochastic actions, but no nondeterministic choices. Let policy computed from δ_1 (δ_2, respectively) be π_1 (respectively, π_2) and let it have success probability p_1 (p_2, respectively). Let this policy have an expected value v_1 (v_2, respectively) given by an accumulated expected reward over the horizon of interest. The standard MDP point of view requires that the optimal policy π corresponding to δ must be chosen between π_1 and π_2 based on their expected values only: the policy with the highest expected value is selected.

On the other hand, from the Golog perspective, if we treat the program as advice, one might want an agent to select the course of action that has the highest probability of successful execution. As an extreme case, suppose the program δ above is given to an agent where (given a specific initial situation) the success probabilities are $p_1 = 0$, $p_2 = 1$, and expected values of policies are $v_1 = 10$, and $v_2 = 0$. While the choice of subprogram δ_2 has lower expected value than δ_1, that choice is more "faithful" to the program. An agent that chooses δ_1 is "refusing to execute" the program δ to completion. This may in fact be a good thing: in this situation, the option δ_2 offered by the programmer is clearly a bad one as it has low expected value. But one may wish the agent to execute the program given whenever possible, in which case option δ_2 *should* be chosen.

The issue is more subtle in less extreme cases (e.g., if v_1 is slightly greater than v_2, but p_1 is slightly less than p_2, what is the optimal execution of δ?). Rather than make a specific design choice, we view this as a multi-criteria objective function that can be optimized however one sees fit: specifically, we assume in the semantics of optimal execution (and the implementation of an interpreter) that a predicate $\leq$ is available that compares pairs of the form $\langle p, v\rangle$, where p is a success probability and v is an expected value. How one defines this predicate will depend crucially on how one interprets the concept of advice embodied in a program. In our implementation, we use a lexicographic preference where $\langle p_1, v_1\rangle < \langle p_2, v_2\rangle$ whenever $p_1 = 0$ and $p_2 > 0$ (so an agent cannot choose a policy that guarantees abnormal termination). If both p_1 and p_2 are zero, or both are greater than zero, than the v-terms are used for comparison. It is important to note that certain multi-attribute preferences could violate the dynamic programming principle, in which case our search procedure would have to be revised. This is not the case with our lexicographic preference.

Finally, we would like to note that there is yet another subtle conflict between the Golog perspective and the MDP perspective in interpreting a DTGolog program. In the dynamic programming, when we choose an optimality criterion, we take a certain horizon of interest; in particular, we assume here that an optimal policy has a finite number H of actions. Thus, H is simply the number of actions that the agent can execute. But from the Golog perspective, it is more natural to talk about *termination* of a Golog program. The program can include not just sequences of actions, but arbitrary compositions of Golog operators. For this reason, the concept of termination of a Golog program is more general. If a Golog program does terminate after a finite number of steps N, then it may happen that the numbers N and H are not equal. For this reason, in the definition of *BestDo* we consider several cases that correspond to possible differences in the values of N and H.[9] More specifically, if $H < N$, then the interpreter does not execute the program any longer once it is considered the first H actions. Informally, this means that H provides a 'hard upper bound' on the number of decisions that the agent is allowed to make. Therefore, even if the program never terminates (e.g., if it is a while-loop with the termination condition such that its truth value depends on a stochastic action and, hence, the termination cannot be guaranteed), the finite horizon H stipulates that only a finite initial execution of this program may determine what is an optimal policy with respect to this program. In this case, we do not use *Stop*, because this constant indicates that the interpreter either cannot execute an action or cannot evaluate a test expression. Instead, we insert *Nil* in the policy to indicate that the horizon decreased down to 0, we no longer need constraints provided by the program and that the construction of a policy was successfully completed. In the opposite case, if $N < H$, then again we use constant *Nil* rather than *Stop* to indicate that remaining $H - N$ actions will be idle actions *Nil* in an absorbing state that will not produce any effect and will lead to zero reward.

Below we define $BestDo$ inductively on the structure of its first argument δ, which is a Golog program. In these definitions, we use an action constant Nil to represent "end-of-program" marker: once the program is executed to the completion by the interpreter, this marker indicates that there is nothing else to be done. The notation π will be overloaded: in most cases, it is used to denote a policy, but it is used also to denote non-deterministic operators; it will be always possible to disambiguate the meaning of this notation from the context.

1. **Zero horizon.**

   BestDo$(\delta, s, 0, \pi, v, pr) \stackrel{def}{=} \pi = Nil \wedge v = reward(s) \wedge pr = 1.$

   Give up on the program δ if the horizon reaches 0. Note that we define the success probability of the policy $\pi = Nil$ as 1. In other words, we do not care what happens after $h = 0$: as far as decision making is concerned, the computation of an optimal policy was successfully completed.

2. **The null program**

   BestDo$(Nil, s, h, \pi, v, pr) \stackrel{def}{=} \pi = Nil \wedge v = reward(s) \wedge pr = 1.$ *Nil* takes the agent into an absorbing state where the agent receives zero reward and remains idle until horizon decreases to 0.

[9] We do not introduce discounting, but this does not mean that the agent can increase accumulated expected reward by simply choosing longer executions because rewards can be negative (and interpreted as costs).

3. **First program action is deterministic.**

$$
\begin{aligned}
&\textit{BestDo}(a;\delta,s,h,\pi,v,pr) \stackrel{def}{=} h>0 \land \\
&\quad \neg \textit{Poss}(a,s) \land \pi = \textit{Stop} \land pr = 0 \land v = \textit{reward}(s) \lor \\
&\quad \textit{Poss}(a,s) \land \\
&\qquad \exists(\pi',v')\, \textit{BestDo}(\delta, do(a,s), h\text{–}1, \pi', v', pr) \land \\
&\qquad\qquad \pi = a\,;\, \pi' \ \land\ v = reward(s) + v' \,.
\end{aligned}
$$

A program that begins with a deterministic agent action a (if a is possible in situation s) has its optimal policy π defined as a followed by the optimal policy π' for the remainder of the program δ in situation $do(a,s)$. Its value is given by the expected value of this continuation plus the reward in s (action cost for a can be included without difficulty), while its success probability $P(\pi)$ is given by the success probability $P(\pi')$ of its continuation. If a is *not* possible at s, the policy is simply the *Stop* action, the success probability is zero, and the value is simply the reward associated with situation s.

4. **First program action is stochastic.**

When a is a stochastic action for which nature selects one of the actions in $\{n_1,\ldots,n_k\}$, where all available nature choices are enumerated (i.e., $choice'(a) \stackrel{def}{=} \{n_1,\ldots,n_k\}$), then

$$
\begin{aligned}
&\textit{BestDo}(a;\delta,s,h,\pi,v,pr) \stackrel{def}{=} h>0 \land \\
&\qquad \exists(\pi',v').\textit{BestDoAux}(choice'(a), a, \delta, s, h-1, \pi', v', pr) \land \\
&\qquad\qquad \pi = a\,;\, \textit{senseEffect}(a)\,;\, \pi' \ \land\ v = reward(s) + v'.
\end{aligned}
$$

The resulting policy is $a; \textit{senseEffect}(a); \pi'$ where π' is the policy delivered by *BestDoAux*. Intuitively, this policy says that the robot should first perform action a, at which point nature will select one of $n_1,\ldots,n_k$ to perform (with probabilities $prob(n_i,a,s)$), then the robot should sense the outcome of action a (which will tell it which n_i nature actually did perform), then it should execute the policy delivered by *BestDoAux*. Recall that $\textit{senseEffect}(a)$ is a domain specific procedure that includes one or a sequence of sense actions. Next, we consider two cases: when $k=1$ and when $k \geq 2$. Suppose $k=1$, i.e. only one nature's action n_k remains to be considered as an outcome of a , then

$$
\begin{aligned}
&\textit{BestDoAux}(\{n_k\}, a, \delta, s, h, \pi, v, pr) \stackrel{def}{=} \\
&\quad \neg \textit{Poss}(n_k,s) \land \textit{senseCond}(n_k,\phi_k) \land \pi = (\phi_k)?\,;\textit{Stop} \ \land v = 0 \land pr = 0 \lor \\
&\quad \textit{Poss}(n_k,s) \land \textit{senseCond}(n_k,\phi_k) \land \\
&\qquad \exists(\pi',v',pr')\, \textit{BestDo}(\delta, do(n_k,s), h, \pi', v', pr') \land \\
&\qquad\qquad \pi = (\phi_k)?\,;\, \pi' \ \land\ v = v' \cdot prob(n_k,a,s) \ \land\ pr = pr' \cdot prob(n_k,a,s).
\end{aligned}
$$

Let ϕ_k be the sense condition for nature's action n_k, meaning that evaluating that ϕ_k

is true is necessary and sufficient for the robot to conclude that nature actually performed action n_k, among the choices $\{n_1, \ldots, n_k\}$ available to her by virtue of the robot having done stochastic action a. When n_k is possible in s, then given the probability $prob(n_k, a, s)$ of this outcome, the optimal policy is the test expression $(\phi_k)?$ followed by the optimal policy π' for the remaining program δ, the value of this policy is the value v' of π' weighted by the probability $prob(n_k, a, s)$ (to compute the expected value v), and the success probability pr of π is determined by multiplying the success probability pr' of π' by $prob(n_k, a, s)$ (occurrences of nature's actions are independent). Otherwise, if n_k is not possible at s, the policy is the *Stop* action that brings zero reward and the success probability is zero. Suppose $k \geq 2$, then

$$
\begin{array}{l}
\mathit{BestDoAux}(\{n_1, \ldots, n_k\}, a, \delta, s, h, \pi, v, pr) \stackrel{def}{=} \\
\quad \neg \mathit{Poss}(n_1, s) \wedge \mathit{BestDoAux}(\{n_2, \ldots, n_k\}, a, \delta, s, h, \pi, v, pr) \vee \\
\quad \mathit{Poss}(n_1, s) \ \wedge \ \exists(\pi', v', pr').\, \mathit{BestDoAux}(\{n_2, \ldots, n_k\}, a, \delta, s, h, \pi', v', pr') \wedge \\
\qquad \exists(\pi_1, v_1, pr_1).\, \mathit{BestDo}(\delta, do(n_1, s), h, \pi_1, v_1, pr_1) \ \wedge \ \mathit{senseCond}(n_1, \phi_1) \wedge \\
\qquad\qquad \pi = \textbf{if } \phi_1 \textbf{ then } \pi_1 \textbf{ else } \pi' \ \wedge \\
\qquad\qquad\qquad v = v_1 \cdot prob(n_1, a, s) + v' \ \wedge \\
\qquad\qquad\qquad\quad pr = pr_1 \cdot prob(n_1, a, s) + pr'.
\end{array}
$$

BestDoAux determines a policy in the form of a conditional plan

$$
\begin{array}{l}
\textbf{if } \ \phi_{i_1} \ \textbf{then } \ pol_1 \ \textbf{else } \ \textbf{if } \ \phi_{i_2} \ \textbf{then } \ pol_2 \cdots \\
\qquad \textbf{else } \ \textbf{if } \ \phi_{i_{m-1}} \ \textbf{then } \ pol_{m-1} \ \textbf{else } \ (\phi_{i_m})?\,;\, pol_m.
\end{array}
$$

Here, $n_{i_1}, \ldots, n_{i_m}$ are all of nature's actions among $\{n_1, \ldots, n_k\}$ that are possible in s, and for each n_{i_j}, pol_j is an optimal policy returned by the *BestDo* interpreter given the program δ, in situation $do(n_{i_j}, s)$. (We use notation pol_j to avoid confusion with π_j that denotes a policy that will be computed for nature's action n_j.)

5. **First program action is a test.**

$$
\begin{array}{l}
\mathit{BestDo}(\phi?; \delta, s, h, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\
\quad \phi[s] \wedge BestDo(\delta, s, h, \pi, v, pr) \ \vee \ \neg\phi[s] \wedge \pi = \mathit{Stop} \wedge pr = 0 \wedge v = reward(s).
\end{array}
$$

6. **First program action is the nondeterministic choice of two programs.**

$$
\begin{array}{l}
\mathit{BestDo}((\delta_1 \mid \delta_2); \gamma, s, h, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\
\quad \exists(\pi_1, v_1, pr_1).\, \mathit{BestDo}(\delta_1; \gamma, s, h, \pi_1, v_1, pr_1) \wedge \\
\quad \exists(\pi_2, v_2, pr_2).\, \mathit{BestDo}(\delta_2; \gamma, s, h, \pi_2, v_2, pr_2) \wedge \\
\qquad (\ (pr_1, v_1) \leq (pr_2, v_2) \wedge \pi = \pi_2 \wedge v = v_2 \wedge pr = pr_2 \ \vee \\
\qquad\qquad (pr_1, v_1) > (pr_2, v_2) \wedge \pi = \pi_1 \wedge v = v_1 \wedge pr = pr_1\ ).
\end{array}
$$

Given the choice between two subprograms δ_1 and δ_2, the optimal policy is determined by that subprogram with optimal execution. Recall that there is some subtlety in the interpretation of a DTGolog program: on the one hand, we wish the interpreter to choose a course of action with maximal expected value; on the other, it should follow the advice provided by the program. Because certain choices may lead to abnormal termination – the $Stop$ action corresponding to an incomplete execution of the program – with varying probabilities, the success probability associated with a policy can be loosely viewed as the degree to which the interpreter adhered to the program. The predicate $\leq$ compares pairs of the form (p, v), where p is a success probability and v is an expected value. [10]

7. **Conditionals.**

$$\textit{BestDo}((\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2); \gamma, s, h, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge$$
$$\phi[s]\ \wedge\ BestDo(\delta_1; \gamma, s, h, \pi, v, pr) \vee$$
$$\neg\phi[s]\ \wedge\ BestDo(\delta_2; \gamma, s, h, \pi, v, pr).$$

Let the program start with a conditional **if** ϕ **then** δ_1 **else** δ_2. If the test expression evaluates to true in s, then the optimal policy must be computed using *then*-branch, otherwise, the optimal policy must be computed following *else*-branch.

8. **Nondeterministic finite choice of action arguments.**

$$\textit{BestDo}((\pi(x : \tau)\delta); \gamma, s, h, pol, v, pr) \stackrel{def}{=} h > 0 \wedge$$
$$\textit{BestDo}(\delta|^x_{c_1}\ \mid\ \cdots\ \mid\ \delta|^x_{c_n}); \gamma, s, h, pol, v, pr)$$

The programming construct $\pi(x : \tau)\delta$ means nondeterministically choose an element x from the finite set $\tau = \{c_1, \ldots, c_n\}$, and for that x, do the program δ. It therefore is an abbreviation for the program $\delta|^x_{c_1}\ \mid\ \cdots\ \mid\ \delta|^x_{c_n}$, where $\delta|^x_c$ means substitute c for all free occurrences of x in δ.

9. **Associate sequential composition to the right.**

$$\textit{BestDo}((\delta_1; \delta_2); \delta_3, s, h, \pi, v, pr) \stackrel{def}{=}\ h > 0\ \wedge \textit{BestDo}(\delta_1; (\delta_2; \delta_3), s, h, \pi, v, pr).$$

This is needed to massage the program to a form in which its first action is one of the forms suitable for application of rules 2-8.

10. **Nondeterministic choice of action arguments.**

$$\textit{BestDo}((\pi\ x)\delta(x); \gamma, s, h, pol, v, pr) \stackrel{def}{=} h > 0 \wedge$$
$$(\exists x).\textit{BestDo}(\delta(x); \gamma, s, h, pol, v, pr)$$

[10]Recall that $(p_1, v_1) < (p_2, v_2)$ whenever $p_1 = 0$ and $p_2 > 0$. If both p_1 and p_2 are zero, or both are greater than zero, than the values of policies are used for comparison.

This is a non-decision-theoretic version of nondeterministic choice: pick an argument and compute an optimal policy given this argument. We need this operator because it will be used to choose moments of time and values returned from sensors.

11. **While-loop is the first program action.**

    This specification requires second order logic; an implementation is provided in Appendix C.1.

12. **Procedure is the first program action.**

    There is also a suitable expansion rule when the first program action is a procedure call. This is similar to the rule for Golog procedures [Levesque *et al.*, 1997, De Giacomo *et al.*, 2000], and requires second order logic to characterize the standard fixed point definition of recursive procedures. An implementation is provided in Appendix C.1.

Finally, it is instructive to compare the semantics of Golog and DTGolog interpreters in terms of interaction between operators. More specifically, it is interesting to observe that the sequential composition and the nondeterministic choice operators are distributive in Golog, but they are not distributive in DTGolog. We continue with Example 5.2.6 to illustrate this. First, if the sequential operator occurs before brackets with nondeterministic choice between tests,

$$flip(1)\ ;\ [\ (head(1))?\ \mid\ (\neg head(1))?\ ]\ ,$$

then this program yields the only one optimal policy

$$flip(1)\,;\ \textit{senseEffect}(flip(1))\,;\ \textbf{if}\ \ head(1)\ \textbf{then}\ \ Nil\ \textbf{else}\ \ (\neg head(1))?\,;\ Nil.$$

The value of this optimal policy is 50 and the probability of its successful termination is 1. Second, if both sides of nondeterministic choice include the sequential composition of the same stochastic action with two different test expressions,

$$[\,flip(1)\,;\ (head(1))?\,]\ \mid\ [\,flip(1)\,;\ (\neg head(1))?\,]$$

then this program yields two different optimal policies

$$flip(1)\,;\ \textit{senseEffect}(flip(1))\,;\ \textbf{if}\ \ head(1)\ \textbf{then}\ \ Nil\ \textbf{else}\ \ Stop\ \ \text{ and}$$

$$flip(1)\,;\ \textit{senseEffect}(flip(1))\,;\ \textbf{if}\ \ head(1)\ \textbf{then}\ \ Stop\ \textbf{else}\ \ (\neg head(1))?\,;\ Nil.$$

The value of each optimal policy is 50, but the probability of successful termination is 0.5 only. If the DTGolog interpreter selects the left branch in the second program and the coin falls heads up, then the test expression can be evaluated to true, but if the coin falls tails up, then the expression evaluates to false and in this case it cannot be executed successfully. Thus, the interpreter will return the first policy in this case, and it will terminate abnormally in 50% of trials. If the DTGolog interpreter selects the right branch in the second program, then, similarly to the previous case, the interpreter will compute the second policy that also will terminate abnormally in 50% of cases. Of course, if the programmer writes a deterministic action before a nondeterministic choice in a program, then it is easy to see that the sequential composition of this action and the nondeterministic choice operators will be distributive (an optimal policy will be the same in both cases) In addition, according to the semantics of Golog, a pair of similar programs will yield identical plans in both cases. This example indicates that in programs intended for DTGolog, if the same sequence of stochastic actions occurs in each branch of a nondeterministic choice operator, then one might wish to rewrite such a program and put the nondeterministic choice after this sequence of stochastic actions.

5.2.3 Computing a Policy and Its Value and Probability

Analogously to the case for Golog, *BestDo*$(prog, s, horiz, pol, val, prob)$ is an *abbreviation* for a situation calculus formula whose intuitive meaning is that pol is an optimal policy resulting from evaluating the program $prog$ beginning in situation s, that val is its value, and $prob$ the probability of a successful execution of this policy. Therefore, given a program δ, and horizon H, one *proves*, using the situation calculus axiomatization of the background domain described in Section 5.2.1, the formula $\exists(pol, val, prob) BestDo(\delta; Nil, S_0, H, pol, val, prob)$. Any binding for pol, val and $prob$ obtained by a constructive proof of this sentence determines the result of the program computation.

A Prolog implementation of DTGolog interpreter is provided in Appendix C.1. To illustrate the functionality of the interpreter, we return to Example 5.2.6 and its implementation in Section C.2.

Example 5.2.6 (continued).

To demonstrate the computational advantages provided by DTGolog framework over unconstrained decision tree search, we consider the following two procedures:

proc search
$(flip(1) \mid flip(2) \mid flip(3) \mid flip(4) \mid flip(5))\ ;\ search$
endProc

proc constr
if $head(1)\ \&\ head(3)\ \&\ head(5)$
then $(flip(2) \mid flip(4))\ ;\ constr$
else $(flip(1) \mid flip(3) \mid flip(5))\ ;\ constr$
endProc

The first procedure *search* does not provide any 'advice' on how to find an optimal policy. The second procedure provides constraints on the search tree: the second and forth coins should be flipped only if the agent is in one of the states where all odd-numbered coins are heads up; otherwise, one of the odd-numbered coins should be flipped first. Suppose that the horizon is 5. Given the program *constr*, the computation of an optimal policy takes 1 sec, and given the program *search* the computation takes 13 seconds, on a laptop computer with 1Gb of memory and 1.6Gz processor running Linux (this example can run with any standard implementation of Prolog). In both cases, the optimal policy is unique, the expected value of the initial state $(0, 0, 0, 0, 0)$ with respect to this policy is 640.625 (according to the rewards and probabilities given in Example 5.2.6) and the probability of successful termination of this policy is 1. To see why unconstrained search takes so much time, it is sufficient to think about the size of the search tree. The procedure *search* commands to execute any of 5 possible actions on each recursive call, and each action leads equiprobably to 2 different states. Therefore, in the case of horizon 5, the procedure compares $O(10^5)$ different branches of the decision tree to compute an optimal policy. For comparison, the size of the search tree explored by the procedure *constr* is determined by actions $flip(1), flip(3), flip(5)$ (each of them leads to 2 different states) followed by remaining two stochastic actions $flip(2)$ and $flip(4)$. In total, the procedure *constr* compares $6^3 \cdot 4$ branches, this number is much less than 10^5 branches in the unconstrained tree.

Thus, this example confirms our expectations that additional constraints on the number of policies that need to be considered will significantly decrease the time required by a decision tree-based search procedure to compute an optimal policy. Note that the state space in this example has only 32 states and any state-of-the art MDP solver will compute an optimal policy almost instantly, but this example clearly illustrates the advantages provided by constraints.

In the subsequent sections, we will demonstrate that there are realistic domains (such as office delivery tasks), where constraints on policies are quite natural and dictated by the structure of the decision problem that must be solved. In Section 5.4, we consider a detailed example of a delivery domain and provide some evidence that in this domain DTGolog has computational advantages over standard dynamic programming algorithms in terms of computation time. In Section 5.5, we compare performance provided by the DTGolog interpreter to performance achieved with SPUDD [Hoey *et al.*, 1999, Hoey *et al.*, 2000] using a well known FACTORY domain as a benchmark for comparison.[11] In Section 5.6, we discuss our experience of using

[11] In [Hoey *et al.*, 1999, Hoey *et al.*, 2000], FACTORY example is represented using an algebraic decision diagram (ADD).

DTGolog for high-level robot programming and argue that DTGolog provide a flexible tool for adapting behavior of the robot to changes in realistic environments. Finally, we review briefly the literature related to macro-actions, and mention several other relevant publications.

However, before we proceed, we would like to consider in the next section an important side issue of how the full observability assumption can be implemented on real robots. In all subsequent sections, we assume that we deal with a fully observable MDP. More specifically, we would like to discuss our representation for sensing actions in robotics context and consider several examples.

5.3 Sensing actions

In Chapter 3, we developed an approach to reasoning about both sensing and physical actions. In this section we would like to show how our approach is used in high-level robot programming.

In contrast to physical actions, sensing actions do not change any properties of the external world, but return values measured by sensors. Despite this difference between them, it is convenient to treat both physical and sensing actions the same in successor state axioms. This approach is justifiable if fluents represent what the robot 'believes' about the world (see Figure 5.1). More specifically, let the high-level control module of the robot be provided with the internal logical model of the world (the set of precondition and successor state axioms) and the axiomatization of the initial situation. The programmer expresses in this axiomatization her (incomplete) knowledge of the initial situation and captures certain important effects of the robot's actions, but some other effects and actions of other agents may remain unmodelled. When the robot does an action in the real world (i.e., the high-level control module sends a command to effectors and effectors execute this command), it does not know directly and immediately what effects in reality this action will produce: the high-level control module may only compute expected effects using the internal logical model. Similarly, if the robot is informed about actions executed by external agents, the high-level control module may compute expected effects of those exogenous actions (if axioms account for them). Thus, from the point of view of the programmer who designed the axioms, the high-level control module maintains the set of 'beliefs' that the robot has about the real world. This set of 'beliefs' needs feedback from the real world because of possible contingencies, incompleteness of the initial information and unobserved or unmodelled actions executed by other agents. To gain the feedback, the high-level control module requests information from sensors and they return requested data.

We find it convenient to represent the information gathered from sensors by an argument of the sensing action: the argument will be instantiated to a particular value at run time when the sensing action is executed. Then, all the high-level control module needs to know is the current situation (action log); the expected effects of actions on (beliefs about) fluents can be computed from the given sequence of physical and sensing actions.

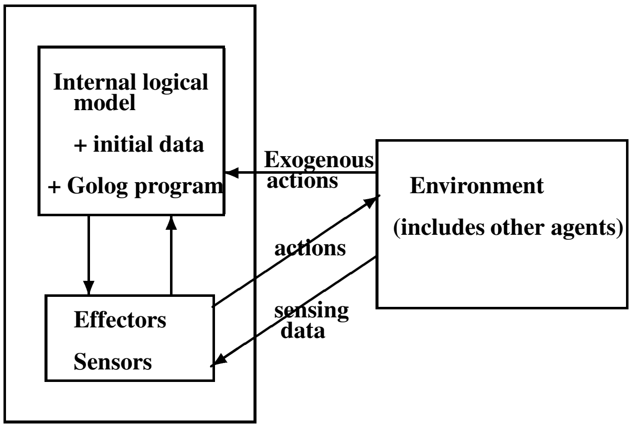

Figure 5.1: A high-level control module and low-level modules interacting with the environment.

In the sequel, we consider only deterministic sensing actions. We suggest representing sensing actions by the functional symbol $sense(q, v, t)$ where q is what to sense, v is term representing a run time value returned by the sensors and t is time. We proceed to consider several examples of successor state axioms that take advantage of our representation.

Example 5.3.1: Let $sense(Coord, l, t)$ be the sensing action that returns the pair $l = (x, y)$ of geometric coordinates of the current robot location on the two-dimensional grid and $gridLoc(x, y, s)$ be a relational fluent that is true if (x, y) are the coordinates of the robot's location in s. In this example we assume that all actions are *deterministic* (we do this only to simplify the exposition of this example). The process of moving (represented by the relational fluent $moving(x_1, y_1, x_2, y_2, s)$) from the grid location (x_1, y_1) to (x_2, y_2) is initiated by the deterministic instantaneous action $startMove(x_1, y_1, x_2, y_2, t)$ and is terminated by the deterministic action $endMove(x_1, y_1, x_2, y_2, t)$. These two actions are under the robot control. The robot may also change its location if it is transported by an external agent from one place to another. The exogenous actions $take(x_1, y_1, t)$ and $put(x_2, y_2, t)$ account for this and the fluent $transported(s)$ represents the process where the robot is transported by an external agent). Of course, the robot does not know when and whether exogenous actions $take(x_1, y_1, t)$ and $put(x_2, y_2, t)$ occur, but if they do, then the robot can find this out by sensing coordinates. In addition, the robot can use its sensing action to update coordinates when it is in the process of moving. The following successor state and precondition axioms characterize the aforementioned fluents and actions.

$$
\begin{aligned}
&gridLoc(x, y, do(a, s)) \equiv \\
&\quad (\exists t, l)\,(a = sense(Coord, l, t) \wedge xCrd(l) = x \wedge yCrd(l) = y) \vee \\
&\qquad (\exists t, x', y')a = \textit{endMove}(x', y', x, y, t) \ \vee\ (\exists t)a = put(x, y, t) \vee \\
&\qquad gridLoc(x, y, s) \wedge \neg(\exists t, x', y')a = put(x', y', t) \wedge \\
&\qquad\quad \neg(\exists l, t)a = sense(Coord, l, t) \wedge \\
&\qquad\quad\ \neg(\exists x, y, x', y', t)a = \textit{endMove}(x, y, x', y', t)\ ,
\end{aligned}
$$

where $xCrd(l), yCrd(l)$ denote, respectively, x and y components of the current location l. This means that the center of the robot is located in (x, y) in the situation that results from doing a in s iff one of the following conditions holds: (a) the last action a that the robot did was sensing its location l and according to the data returned by sensors the current coordinates are x and y; (b) the robot was moving from a location (x', y') to (x, y), and at time t it stopped moving, or (c) in the previous situation the robot was being transported by another agent and at certain time t that agent does the action a of placing the robot at (x, y), or (d) in the previous situation the center of the robot was already located in (x, y) and the most recent action a was neither to put the robot somewhere, nor sense coordinates, nor end moving at a different point.

$$
\begin{aligned}
&moving(x_1, y_1, x_2, y_2, do(a, s)) \equiv\ (\exists t)a = \textit{startMove}(x_1, y_1, x_2, y_2, t) \vee \\
&\qquad moving(x_1, y_1, x_2, y_2, s) \wedge \neg(\exists t)a = \textit{endMove}(x_1, y_1, x_2, y_2, t).
\end{aligned}
$$

$$
\begin{aligned}
&transported(do(a, s)) \equiv\ (\exists t, x, y)\, a = take(x, y, t) \vee \\
&\qquad transported(s) \wedge \neg(\exists t, x', y')a = put(x', y', t) \vee \\
&\qquad\quad (\exists x, y, t, l)\,[\, gridLoc(x, y, s) \wedge \neg\exists x', y' moving(x, y, x', y', s) \wedge \\
&\qquad\qquad a = sense(Coord, l, t) \wedge (xCrd(l) \neq x \vee yCrd(l) \neq y)\,].
\end{aligned}
$$

This says that the robot is transported by an external agent if there exists time t at which the robot was taken by the agent from certain point (x, y) or the robot was transported by someone in the previous situation and that agent did not put the robot somewhere at some time t'. In addition, the last two lines allow the robot to determine from sensing that it is transported by someone even if there is no occurrence of the $take(x, y, t)$ action in s. In other words, even if the robot does not know that an external agent took it and started to transport to elsewhere, the robot can find this out. More specifically, if the current location of the robot is at (x, y), the robot is not moving, and the last action is the action of sensing coordinates such that sensors tell the robot that its location is different from (x, y), then the fluent $transported$ is true in the situation resulting from execution of this action in s. According to the next axiom, the robot can start moving from (x_1, y_1) to (x_2, y_2) at the moment of time t in the situation s, if the robot is not transported by anybody, and in addition, if it not moving between any pair of locations, and the current location of the center of the robot is the point (x_1, y_1).

$$\begin{aligned} &\mathit{Poss}(\mathit{startMove}(x_1, y_1, x_2, y_2, t), s) \equiv \neg transported(s) \wedge \\ &\qquad \neg(\exists x, y, x', y')\, moving(x, y, x', y', s) \wedge gridLoc(x_1, y_1, s), \\ &\mathit{Poss}(\mathit{endMove}(x_1, y_1, x_2, y_2, t), s) \equiv moving(x_1, y_1, x_2, y_2, s), \end{aligned}$$

$$\begin{aligned} &\mathit{Poss}(take(x, y, t), s) \equiv \neg transported(s) \wedge gridLoc(x, y, s) \\ &\mathit{Poss}(put(x, y, t), s) \equiv transported(s). \end{aligned}$$

Informally speaking, the very last axiom is saying that at the moment t an external agent can put the robot at a point (x, y) in the situation s if the robot is transported (by that agent) in s. The previous axiom means that the action $take(x, y, t)$ is possible in s, if in this situation the robot is not transported by someone, and the center of the robot is located in (x, y).

As usual, our basic action theory must include the unique name axioms (but we omit them). Imagine that in the initial situation the robot stays at (0,0). Later, at time 1, it starts moving from (0,0) to (2,3), but when the robot stops at time 11 and senses its coordinates at time 12, its sensors tell it that it is located at (4,4). The sensory information is inconsistent with the expected location of the robot, but this discrepancy can be attributed to unobserved actions of an external agent who transported the robot. An alternative explanation, that the robot moved again would not work, because both *startMove*, *endMove* actions are under the robot control and if they occur, then they both will be mentioned in the robot's situation term. Hence, the final situation is not

$$do(sense(Coord, (4, 4), 12), do(\mathit{endMove}(0, 0, 2, 3, 11), do(\mathit{startMove}(0, 0, 2, 3, 1), S_0))),$$

as we originally expected, but the situation resulting from the execution of exogenous actions $take(2, 3, T_1)$ and $put(4, 4, T_2)$, in the situation when the robot ends moving, followed by the sensing action. The exogenous actions occurred at unknown times T_1, T_2 (we may say about the actual history only that $11 \leq T_1 < T_2 \leq 12$).[12]

[12]In the sequel, we do not consider how the discrepancy between results of a deterministic action and observations obtained from sensors can be explained by occurrences of unobserved exogenous actions. However, the inconsistency between physical and sensory actions can be resolved by solving a corresponding diagnostic problem (e.g., see [McIlraith, 1998]).

Example 5.3.2: Here we consider a stochastic $endGo$ action: when the robot does it, nature can choose one of the two outcomes: the robot arrives successfully or fails to arrive (gets lost in transition). The robot can determine its location using data from sonar and laser sensors. But if the last action was not sensing coordinates (x, y), then the current location can be determined from the previous location and the actions that the robot has executed. The process of going from l to l' is initiated by the deterministic action $startGo(l, l', t)$ and is terminated by the *stochastic* action $endGo(l, l', t)$ (axiomatized in Section 5.4). The robot is located in l if and only if (1) sensors determined that coordinates of the robot are (x, y) and the point (x, y) is inside of the location l, or (2) the last action is arrive at l, or (3) the robot is located in l and the last action is not such that the robot should be at another location:

$$
\begin{aligned}
robotLoc(l, do(a, s)) \equiv\ & (\exists t, v, x, y)a = sense(Coord, v, t) \wedge \\
& xCoord(v) = x \wedge yCoord(v) = y \wedge inside(l, x, y) \vee \\
& (\exists t, l_1)\, a = endGoS(l_1, l, t) \ \vee (\exists t, l_1)\, a = endGoF(l_1, l, t) \vee \\
& robotLoc(l, s) \wedge \neg(\exists l_2, l_3, t)a = endGoS(l_2, l_3, t) \wedge \neg(\exists l_2, l_3, t)(a = endGoF(l_2, l_3, t)) \wedge \\
& \neg(\exists t', v', x', y', l')\, [\, a = sense(Coord, v', t') \wedge \\
& xCoord(v') = x' \wedge yCoord(v') = y' \wedge inside(l, x', y') \wedge l \neq l'\,],
\end{aligned}
$$

where the predicate *inOffice*(l, x, y) is true if the pair (x, y) is inside of the office l of an employee. We assume that all coordinates (x', y') which are not inside offices of employees are inside the hall, where the robot can get stuck when it is moving between offices.

When the robot stops and senses coordinates, it determines its real location. Subsequently, the high-level control module can identify the outcome of the stochastic action $endGo(l, l', t)$: whether the robot stopped successfully or failed to arrive at the intended destination. This can be achieved by evaluating appropriate sense conditions defined in

senseCond$(endGoS(l_1, l_2, t), robotLoc(l_2))$ and

senseCond$(endGoF(l_1, l_2, t), robotLoc(Hall))$

as we explained in Section 5.2.1, when we discussed the background basic action theory that represents an MDP. More specifically, in the case if nature chooses $endGoS(l, l', t)$, sensors find that the robot is located in l', and the expression $robotLoc(l')$ evaluates to true in the resulting situation. This tells the robot that it is located in a state where it was expected to arrive (as a positive side-effect, the robot also learns what action nature selected). In the case, if nature chooses $endGoF(l, Hall, t)$, when the robot was actually moving to the office l' of an employee, the expression $robotLoc(Hall)$ evaluates to true in the situation that results from doing a sense action and the robot learns which action was selected by nature. Of course, in a general case, more than one sensing actions can be required and sense conditions can be arbitrary complex situation-suppressed expressions that needs to be evaluated before the robot can find which of many available choices was actually executed by nature in a previous situation.

Example 5.3.3: Let $give(item, pers, t)$ be a stochastic action that has two different outcomes: $giveS(item, pers, t)$ – the robot gives successfully an $item$ to $pers$ – and $giveF(item, pers, t)$ – the action of giving an $item$ to $pers$ is unsuccessful. Let $sense(Ackn(item), v, t)$ be the action of sensing whether delivery of the $item$ to $person$ was successful or not: if it was, then delivery is acknowledged and v=1, if not – the result of sensing is v=0.[13] The following successor state axiom characterizes how the fluent $hasCoffee(person, s)$ changes from situation to situation: whenever the robot is in the office of $person$ and it senses that one of its buttons was pressed, it is assumed that the $person$ pressed a button to acknowledge that she has a coffee. From this sensory information, the high-level control module can identify whether the outcome of $give(item, person, t)$ was successful or not.

$$\begin{aligned}
&hasCoffee(pers, do(a, s)) \equiv\ (\exists t)a = giveS(\textit{Coffee}, pers, t) \vee \\
&\qquad robotLoc(office(pers), s) \wedge (\exists t)a = sense(Ackn(\textit{Coffee}), 1, t) \vee \\
&\qquad hasCoffee(pers, s) \wedge \\
&\qquad\qquad [\, (\neg\exists t)a = sense(Ackn(\textit{Coffee}), 0, t) \wedge (\neg\exists t')a = giveF(\textit{Coffee}, pers, t')\,]
\end{aligned}$$

A person has coffee in the next situation, if she had a coffee in the previous situation and the last action executed by the robot was not the action of sensing that returned a signal 0 from buttons and not nature's choice meaning that the stochastic agent action of giving an item failed. This axiom assumes that the robot can attempt many deliveries to a person (presumably, because the person requests more coffee), some of them can lead to giving a coffee to the person successfully, but once the stochastic $give(item, pers, t)$ action fails, this results in a situation where the person has no coffee (independently of the past).

Example 5.3.4: The battery voltage remains constant if the battery is connected to the charger, otherwise the voltage varies unpredictably and the high-level control module of the robot can determine it only by doing the measurement $sense(Battery, x, time)$: this sensing action returns the real value x of the voltage. We use the relational fluent $connected(s)$ to assert that the

[13]As we mentioned in Section 2.5, the stochastic action $give(item, person, t)$ can be implemented as a verbal announcement that the robot arrived with an $item$ addressed to $person$. Also, acknowledgment v=1 can be implemented if a recipient presses one of robot's buttons during a certain interval of time; if no button was pressed, then v=0.

battery is connected to the charger (We omit the successor state axiom for this fluent because it is not essential for this example.)

$$voltage(x, do(a, s)) \equiv connected(s) \land x = 24 \lor \\ \neg connected(s) \land (\exists t) a = sense(Battery, x, t).$$

Example 5.3.5: Assume that the robot has no devices to sense its location directly, but can measure its velocity (e.g., by counting revolutions of its motors): we model this by the action $sense(Vel, v, t)$. For simplicity of presentation, we assume that the motion is uniform and the robot moves along the x axis; we also assume in this example that both *startMove* and *endMove* actions are deterministic (their precondition axioms are similar to axioms in the example 1). In this example, when the robot starts moving it moves with a certain velocity that remains constant but unknown, unless the robot measures its velocity.

Given the value returned by the velocity sensor, we are interested in computing the current location of the robot:[14]

$$moving(do(a, s)) \equiv (\exists t) a = \textit{startMove}(t) \lor moving(s) \land \neg(\exists t) a = \textit{endMove}(t).$$

$$velocity(v, do(a, s)) \equiv (\exists t) a = sense(Vel, v, t) \lor (\exists t) a = \textit{endMove}(t) \land v = 0 \lor \\ velocity(v, s) \land (\neg \exists t) a = \textit{endMove}(t) \land (\neg \exists t, v) a = sense(Vel, v, t).$$

Note that according to the last axiom, if the robot ended to move in the past and is not moving in the current situation s, then its velocity 0. In addition, when the robot starts moving, its velocity remains 0 because the action *startMove*(t) has no effect on the velocity, but if the robot measures velocity in the process of moving soon after starting to move, then it gets to know the actual velocity.

$$\begin{aligned} & locat(x, do(a, s)) \equiv \\ & \quad (\exists v, t_1, t_2, x')\,(a = sense(Vel, v, t_1) \land locat(x', s) \land \\ & \quad\quad start(s) = t_2 \land x = v \cdot (t_1 - t_2) + x') \lor \\ & \quad (\exists v, t_1, t_2, x')\,(a = \textit{endMove}(t_1) \land velocity(v, s) \land \\ & \quad\quad locat(x', s) \land start(s) = t_2 \land x = v \cdot (t_1 - t_2) + x') \lor \\ & \quad (\neg \exists t) a = \textit{endMove}(t) \land (\neg \exists t, v)\, a = sense(Vel, v, t) \land \\ & \quad\quad [\neg moving(s) \land locat(x, s) \lor moving(s) \land \\ & \quad\quad (\exists v, t_1, t_2, x')\,(velocity(v, s) \land locat(x', s) \land \\ & \quad\quad start(s) = t_2 \land time(a) = t_1 \land x = v \cdot (t_1 - t_2) + x')\,]. \end{aligned}$$

[14]Recall that we use the function symbol $start(s)$ to denote the starting time of the situation s and the function symbol $time(a)$ to denote the time when the action a occurs, see Section 2.1.2 for details.

Note that the robot using this axiom may underestimate the change in location, since it is assumed that *startMove*(t) leaves velocity at 0, but the sooner the robot will sense its real velocity in the process of moving, the smaller will be the error in location. Because the 'persistence' part of this last axiom applies to any action different from sensing and stopping, the robot can update its location after executing any other action by using the time argument when this action was executed.

This set of examples demonstrates how sense actions can be used together with physical actions in successor state axioms. As we have mentioned in Chapter 3, with our approach to representation of sense actions it is surprisingly straightforward to use regression to solve the forward projection task. Let $\mathcal{D}$ be a basic action theory defined in Chapter 2 and let $\phi(s)$ be an uniform situation calculus formula. Suppose we are given a ground situational term S that may mention both physical and sensing actions. Thanks to representation introduced above, we can use regression to solve the forward projection task also in the case when S mentions sensing actions explicitly. The remarkable consequence of this fact is that we can use regression to evaluate tests (ϕ)? in Golog programs with sensing actions: no modifications are required to solve the entailment problem whether $\mathcal{D} \models \phi(S)$.

5.4 An Application of DTGolog: A Delivery Example

In this section we describe a simple, but realistic domain where natural constraints on robot behavior can be conveniently expressed in a Golog program. In domains of this type, we can use the *BestDo* interpreter to find an optimal policy for the corresponding MDP.

Imagine a robot that has to deliver coffee and/or mail from the main office where a coffee machine and mail boxes are located. Mail arrives once in the morning and all employees make coffee request once in the morning only by specifying intervals of time when they would prefer coffee be delivered to them. In the main office (abbreviated as MO), a person can put a cup of coffee or mail packages on the robot and we model this by saying that the robot executes the deterministic action *pickup*$(item, t)$. The process of going from a location l_1 to l_2 is initiated by a deterministic action *startGo*(l_1, l_2, t); the robot can go only to offices of employees or to the main office. But the terminating action *endGo*(l_1, l_2, t) is stochastic (e.g., the robot may end up in some location other than an office l_2, say, the hallway). We give nature two choices, *endGoS*(l_1, l_2, t) (successful arrival) and *endGoF*$(l_1, Hall, t)$ (end with failure), and include abbreviations such as

$$choice'(endGo(l_1, l_2, t)) \stackrel{def}{=} \{endGoS(l_1, l_2, t), s), endGoF(l_1, Hall, t)\} \text{ and}$$
$$prob(endGoS(l_1, l_2, t), endGo(l_1, l_2, t), s) \stackrel{def}{=} 0.99$$

(i.e., successful movement occurs with probability 0.99 in any situation). When the robot arrives at the door of an employee, it asks the employee to take delivered items; we model this by the stochastic action $give(item, person, t)$. If the employee is in her office, she takes the item, and presses one of the buttons on the robot, as we discussed in Example 5.3.3, then this action has a successful outcome $giveS(item, person, t)$. But if the employee is not there and the door is closed, nobody acknowledges delivery of the item, then $give(item, person, t)$ fails, reflected by outcome $giveF(item, person, t)$.

This domain is modeled using fluents such as $going(l_1, l_2, s)$, $robotLoc(l, s)$, $carrying(item, s)$, $hasCoffee(person, s)$, etc., that we have already considered in Example 2.1.2. Several other predicates such as $hasMail(person, s)$, $wantsCoffee(person, t_1, t_2, s)$ (person wants coffee at some point in the time period $[t_1, t_2]$), $mailPresent(person, n, s)$ (n is the number of mail packages addressed to $person$), function symbols and constants (e.g., names of people, items and locations) are also used similarly to that example.

The following precondition and successor state axioms characterize these fluents and the actions (sensing actions are always possible).

$$\begin{aligned}
&Poss(startGo(loc_1, loc_2, t), s) \equiv \neg(\exists l, l')going(l, l', s) \wedge \\
&\qquad\qquad robotLoc(loc_1, s) \wedge \exists p(loc2 = office(p)), \\
&Poss(endGoS((loc_1, loc_2, t), s) \equiv going(loc_1, loc_2, s), \\
&Poss(endGoF(loc_1, loc_2, t), s) \equiv \exists loc'(\, going(loc_1, loc', s) \wedge loc' \neq loc_2\,), \\
&Poss(pickup(item, t), s) \equiv item = Coffee \wedge \neg carrying(item, s) \wedge robotLoc(MO, s),
\end{aligned}$$

$$\begin{aligned}
&Poss(pickup(item, t), s) \equiv item = mailTo(pers) \wedge \neg carrying(item, s) \wedge robotLoc(MO, s) \wedge \\
&\qquad \exists n(\, mailPresent(pers, n, s) \wedge n > 0\,), \\
&Poss(putBack(item, t), s) \equiv carrying(item, s) \wedge robotLoc(MO, s), \\
&Poss(giveS(item, person, t), s) \equiv carrying(item, s) \wedge \\
&\qquad robotLoc(office(person), s) \wedge (item = Coffee \vee item = mailTo(person)\,), \\
&Poss(giveF(item, person, t), s) \equiv carrying(item, s) \wedge \\
&\qquad robotLoc(office(person), s) \wedge (item = Coffee \vee item = mailTo(person)\,).
\end{aligned}$$

$$
\begin{aligned}
robotLoc(l, do(a,s)) \equiv\ & (\exists t, v, x, y) a = sense(Coord, v, t) \wedge \\
& xCoord(v) = x \wedge yCoord(v) = y \wedge inside(l, x, y) \vee \\
& (\exists t, l_1)\, a = endGoS(l_1, l, t) \vee (\exists t, l_1)\, a = endGoF(l_1, l, t) \vee \\
& robotLoc(l, s) \wedge \neg(\exists l_2, l_3, t) a = endGoS(l_2, l_3, t) \wedge \neg(\exists l_2, l_3, t)(a = endGoF(l_2, l_3, t)) \wedge \\
& \neg(\exists t', v', x', y', l')\, [\, a = sense(Coord, v', t') \wedge \\
& xCoord(v') = x' \wedge yCoord(v') = y' \wedge inside(l, x', y') \wedge l \neq l' \,],
\end{aligned}
$$

Ignoring actions related to sensing (see Example 5.3.2), this axiom is saying that the robot's location is loc in the situation that results from doing an action a in the previous situation s if the robot's location was loc in s and action a did not change it. Otherwise, if a is either *endGoS* or *endGoF* action, then the robot location is where the robot got to go. Note that this axiom and all remaining axioms do not mention stochastic agent actions: they mention only the corresponding nature's deterministic actions and deterministic agent actions.

$$
\begin{aligned}
& going(l, l', do(a,s)) \equiv (\exists t) a = startGo(l, l', t) \vee \\
& \qquad going(l, l', s) \wedge \neg(\exists t) a = \mathbf{endGoS}(l, l', t) \wedge \neg(\exists t, l''') a = \mathbf{endGoF}(l, l''', t), \\
& wantsCoffee(p, t_1, t_2, do(a,s)) \equiv wantsCoffee(p, t_1, t_2, s) \wedge (\neg\exists t) a = giveS(\mathbf{Coffee}, p, t), \\
& mailPresent(person, 1, do(a,s)) \equiv (\exists t)\, a = putBack(mailTo(person), t) \vee \\
& \quad \exists n. mailPresent(person, n, s) \wedge (\neg\exists t) a = pickup(mailTo(person), person, t),
\end{aligned}
$$

$$
\begin{aligned}
& hasCoffee(pers, do(a,s)) \equiv (\exists t) a = giveS(\mathbf{Coffee}, pers, t) \vee \\
& \quad robotLoc(office(pers), s) \wedge (\exists t) a = sense(Ackn, 1, t) \vee hasCoffee(pers, s), \\
& hasMail(pers, do(a,s)) \equiv (\exists t) a = giveS(mailTo(pers), pers, t) \vee \\
& \quad robotLoc(office(pers), s) \wedge (\exists t) a = sense(Ackn, 1, t) \vee hasMail(pers, s), \\
& carrying(item, do(a,s)) \equiv (\exists t) a = pickup(item, t) \vee \\
& \qquad carrying(item, s) \wedge \neg(\exists p, t)\, a = giveS(item, p, t).
\end{aligned}
$$

The last axiom is saying that the robot is carrying an $item$ in the situation $do(a,s)$ that results from doing an action a in the previous situation s if and only if the robot picked an item at moment t or if the robot was carrying $item$ in s and it did not give successfully the item to a person p (e.g., if the door of p's office is closed; in this case, the item remains on a tray that the robot is carrying). Two previous axioms are saying that a person has an item in $do(a,s)$ if the last action is give this item to the person successfully (this is nature's action) or if the sensor returns the signal 1 (when delivery is acknowledged) or if the person had this item in the previous situation s (once delivered, the item remains with the person who requested this item in the morning).

The following sense conditions distinguish successful from unsuccessful outcomes:

$$
\begin{aligned}
&\textbf{\textit{senseCond}}(n,\phi) \stackrel{def}{=} \\
&\quad \exists l_1, l_2, t(\, n = \textbf{\textit{endGoS}}((l_1, l_2, t) \wedge \phi = robotLoc(l_2) \vee \\
&\qquad\qquad n = \textbf{\textit{endGoF}}(l_1, l_2, t) \wedge \phi = robotLoc(Hall)\,) \vee \\
&\quad \exists person, t(\, n = giveS(Mail, person, t) \wedge \phi = hasMail(person) \vee \\
&\qquad\qquad n = giveS(\textbf{\textit{Coffee}}, person, t) \wedge \phi = hasCoffee(person)) \vee \\
&\quad \exists person, t(\, n = giveF(Mail, person, t) \wedge \phi = (\neg hasMail(person)) \vee \\
&\qquad\qquad n = giveF(\textbf{\textit{Coffee}}, person, t) \wedge \phi = (\neg hasCoffee(person))\,).
\end{aligned}
$$

Probabilities of outcomes are defined by the following abbreviations:

$$
\begin{aligned}
&prob(n, \textbf{\textit{endGo}}(l_1, l_2, t), s) = p \stackrel{def}{=} (\, n = \textbf{\textit{endGoS}}(l_1, l_2, t) \wedge p = 0.99 \vee \\
&\qquad\qquad n = \textbf{\textit{endGoF}}(l_1, l_2, t) \wedge p = 0.01\,). \\
&prob(n, give(item, person, t), s) = p \stackrel{def}{=} (\, n = giveS(item, person, t) \wedge p = 0.95 \vee \\
&\qquad\qquad n = giveF(item, person, t) \wedge p = 0.05\,).
\end{aligned}
$$

There are two *senseEffect*(A) procedures in this application domain that correspond to two stochastic agent actions.

proc *senseEffect*$(endGo(l_1, l_2, t))$ $\;(\pi v, t).\ sense(Coord, v, t)$ **endProc**

proc *senseEffect*$(give(item, person, t))$ $\;(\pi v, t).\ sense(Ackn, v, t)$ **endProc**

Each of these two procedures commands to execute one sensing action with arguments that gets bound to values returned at run time. The $BestDo$ interpreter inserts a *senseEffect*(A) procedure into a policy after each occurrence of a stochastic action A to implement the full observability assumption. According to Definitions 5.2.1 and 5.2.2, of a policy and branches in a policy, and according to the specification of the $BestDo$ interpreter, each occurrence of *senseEffect*(A) in a policy is followed by *if-then-else* operator with sense conditions corresponding to all nature choices that are possible in the current situation. Because each sensing action is added to the current situation with arguments grounded at run time, the robot executing the policy can determine the state uniquely (as well as a deterministic action chosen by nature) by evaluating these sense conditions in the situation that results from executing a sensing action.

The reward for doing the action $giveS(mail, \textit{person}, t)$ to some *person* who does not have mail is the maximum (with respect to time constraints) of $(10n - \frac{t}{10})$, where n - number of mails. Given a situation term corresponding to any branch of the situation tree, it is straightforward to maximize value with respect to choice of temporal arguments assigned to actions in the sequence. (This is a standard linear programming problem subject to constraints on how much time the robot spends on traveling.) The reward for giving a coffee successfully to a person is also the maximum of a linear function of time. If p wants a coffee from t_1 to t_2, then the robot gets the highest reward equal to $\frac{t_2-t_1}{2}$ when it gives coffee at t_1. The reward decreases linearly and equals 0 at moments ($\frac{t_2-t_1}{2}$ - t_1) and t_2; the reward is negative outside the interval $(\frac{t_2-t_1}{2} - t_1, t_2)$.[15] For simplicity, the rewards (costs) for doing all other actions are 0:

$$
\begin{aligned}
&\mathit{reward}(do(a,s)) = v \overset{def}{=} \exists p, t, t_1, t_2 \\
&\qquad (a = giveS(\mathit{Coffee}, p, t) \wedge wantsCoffee(p, t_1, t_2, s) \wedge \\
&\qquad\qquad v \leq \tfrac{t_2-t}{2} \ \wedge \ v \leq t - \tfrac{3*t_1-t_2}{2}\ ) \vee \\
&\qquad (\exists person, t, numb)\ (a = giveS(mail(person), person, t) \wedge \\
&\qquad\qquad mailPresent(person, numb, s) \wedge v = (10 \cdot numb - \tfrac{t}{10}).
\end{aligned}
$$

Suppose that in the initial situation, the robot is located in the main office, one employee Ann wants coffee at some point in the time period [100,150], there are four mail packages addressed to Joe, and a third employee Bill has no requests. From the main office, it takes the robot 45 units of time to travel to Ann's office, 100 units of time to travel to Joe's office, and 20 units of time to travel to the hall (where the robot may get stuck); in addition, it takes 55 units of time to travel from Ann's office to Joe's office. If the robot picks up mail, goes to Joe's office (the probability of reaching it successfully is 0.99) and successfully gives him mail (with the probability 0.95) at time 100, then it would receive the reward 28.215. If the robot would pickup the coffee instead, go to the office of Ann at time 55 and give her the coffee at time 100, then it would receive the reward 23.5125 (the success probabilities for the corresponding actions are the same). If the horizon is 5, then the optimal policy for the robot is to deliver mail to Joe.

The following Golog procedures express natural constraints of the robot's behavior (with abbreviation MO - main office); in the procedure *deliver* we use *people* to denote the finite range: *people* $= \{Bill, Ann, Joe\}$.

proc $deliver$

$\pi(p : \mathit{people})\ \pi t.[deliverCoffee(p,t)]) \ \mid \quad \pi(p : \mathit{people})\ \pi t.[\mathit{deliverMail}(p,t)]),$

endProc

[15] Of course, any monotone function of time will be as good as the linear function that we use here, but this linear function leads to a particularly simple implementation.

where $\pi\, t$ is an operator that chooses a moment of time that can be grounded when the policy will be executed in reality.

proc $deliverCoffee(p, t)$
  if $robotLoc(MO)$ **then**
    if $carrying(\textit{Coffee})$ **then** $serveCoffee(p, t)$
      else $pickup(\textit{Coffee}, t)$; $serveCoffee(p, t)$
  else **if** $(-carrying(\textit{Coffee}))$ **then** $goto(MO, t)$;
  $\pi t_1\, [?(now(t_1))$; $pickup(\textit{Coffee}, t_1)$; $serveCoffee(p, t_1)]$
      else $serveCoffee(p, t)$
endProc

proc $serveCoffee(p, t)$
  if $robotLoc(office(p))$ **then** $give(\textit{Coffee}, p, t)$
    else $goto(office(p), t)$; $\pi t_1[?(now(t_1))$; $give(\textit{Coffee}, p, t_1)]$
endProc

The procedures *serveMail*(p, t) and *deliverMail*(p, t) have identical control structure:

proc *deliverMail*(p, t)
  if $robotLoc(MO)$ **then**
    if $carrying(mailTo(p))$ **then** *serveMail*(p, t)
      else $pickup(mailTo(p), t)$; *serveMail*(p, t)
  else **if** $(-carrying(mailTo(p)))$ **then** $goto(MO, t)$;
  $\pi t_1\, [?(now(t_1))$; $pickup(mailTo(p), t_1)$; *serveMail*$(p, t_1)]$
      else *serveMail*(p, t)
endProc

proc *serveMail*(p, t)
  if $robotLoc(office(p))$ **then** $give(mailTo(p), p, t)$
    else $goto(office(p), t)$; $\pi t_1[?(now(t_1))$; $give(mailTo(p), p, t_1)]$
endProc

The procedures *goto*(l_2, t) and *goBetween* are the same as in Example 2.1.2.

A Prolog implementation of this delivery domain including all Golog procedures mentioned above is provided in Appendix C.6. Charged with executing procedure *deliver*, the *BestDo* interpreter returns the following optimal policy (assume that the horizon equals 5):

```
pickup(mailTo(joe),0) :
  startGo(mo,office(joe),0) : endGo(mo,office(joe),100) :
    if(robotLoc(office(joe)), /* THEN */ give(mailTo(joe),joe,100) :
```

```
          if(hasMail(joe), nil, ?(-(hasMail(joe))) : nil),
    /* ELSE */ ?(robotLoc(hall)) : give(mailTo(joe),joe,20) : stop)
```

The expected utility of this policy is 28.215 and the probability of successful termination is 0.99. It takes 0.02 seconds to compute this policy.[16] Note that the stochastic action $give(mailTo(Joe), Joe, 20)$ is followed by *Stop* in this policy, because both outcomes of the action are not possible in the situation when the robot got stuck in the hall. This conforms with the specification of BestDo given in the previous section (see also the precondition axioms for *giveS* and *giveF*).

We contrast the optimal policy above with the optimal policy that *BestDo* interpreter would compute if it were given the Golog program $\pi(p : \textit{people})\ \pi t.[deliverCoffee(p, t)])$:

```
pickup(coffee, 55) :
  startGo(mo,office(ann),55) : endGo(mo,office(ann),100) :
    if(robotLoc(of(ann)), /* THEN */ give(coffee,ann,100) :
           if(hasCoffee(ann), nil, ?(-(hasCoffee(ann))) : nil),
      /* ELSE */ ?(robotLoc(hall)) : give(coffee,ann,75) : stop)
```

The expected utility of this policy is 23.5125, the total probability of successful termination is 0.99. The computation time is 0.02 seconds.

It is important to compare policies computed by the $BestDo$ interpreter with optimal policies that are computed by exact algorithms. The underlying MDP for this relatively simple domain grows rapidly as the number of people requiring deliveries increases. The state space is characterized by ground fluents such as *mailPresent*$(\textit{person}, n, s)$, *robotLoc*(loc, s), $hasMail(\textit{person}, s)$, and so on. Even restricting the MDP to one piece (or bundle) of mail per person, and *ignoring* the temporal aspect of the problem), in a domain with P people, our MDP has a state space of size $O(2^{5P} \cdot P^3)$ when formulated in the most appropriate way. Indeed, the robot can be in any of $(P + 2)$ locations (P employees, hall and the main office), it can be going between any pair of locations (these contribute $O(P^3)$ different ground instances of $robotLoc$ and $going$), any subset of people can request coffee (giving the factor 2^P), any subset of mail packages can be present (giving the factor 2^P), the robot may carry nothing or carry any subset of packages addressed to any subset of people (this contributes the factor 2^P) and any subset of P people has or does not have mail and has or does not have coffee (this contributes the factor $2^P \cdot 2^P$). The state space complexity, $32^P \cdot P^3$, grows exponentially in

[16] All programs mentioned in this chapter run on the same laptop computer using Linux with 1Gb of RAM and 1.6Gz processor to provide a fair comparison of computation time.

P, and in the case considered above, when $P = 3$ we have about 10^6 states. To compute an exact optimal policy for this MDP we run SPUDD using a straight-forward translation of this domain into algebraic decision diagrams with the following simplifications. Because *Bill* has no requests in our example and our primary interest is an optimal policy that can be executed from the initial state that corresponds to S_0, we consider a subset of the original state space and restrict our attention to ground instances of fluents and actions only for two people: *Joe, Ann*. Instead of two primitive actions $startGo$ and $endGo$, only one primitive action $goto(office(p))$ is considered. These simplifications can only make the task of computing an optimal policy for SPUDD easier and effectively reduce the number of states down to about $8 \cdot 10^3$ states and reduce the number of ground instances of actions for SPUDD. In total, our representation has 11 binary-valued features, 1 multi-valued feature (robot location) and 13 primitive actions (see Appendix C.4 for a copy of the ADD-based representation of this example.) The computation of an optimal policy for this MDP with horizon 5 takes 0.1 seconds which is longer in comparison with 0.02 seconds required for $BestDo$ to compute a policy optimal with respect to given constraints expressed by the Golog program considered above. In the case of infinite horizon, the value of the initial state is 88.1 and an optimal policy commands to load both mail and coffee, go to Ann, give her coffee, then go to Joe and give him mail.

Notice that the Golog program restricts the policy that the robot can implement, leaving only one choice (the choice of person to whom to deliver an item) open to the robot and the rest of the robot's behavior is fixed by the program. Furthermore, programs of this type are, arguably, quite natural. However, it is important to note that there is a price to pay if we want to use this program when we search for an optimal policy in the decision tree. Indeed, structuring a program this way may, in general, preclude optimal behavior. For instance, by restricting the robot to serving one person at a time, carrying two items at once won't be considered. If Ann and Joe share an office and the horizon is 7, then an obvious optimal policy is pickup mail (assume that the robot can pick up all packages at once), pickup coffee at 55, go to the office and give delivered items to Ann and Joe at 100. Hence, the robot loses almost half of its accumulated expected reward because the program does not allow us to consider more flexible policies. But in circumstances where carrying several items is impossible, the programmer allows optimal behavior and describes for the robot how to deliver an item, leaving for the robot to decide only on the order of deliveries. Even where overlapping deliveries may be optimal, the "nonoverlapping" program may have sufficiently high utility that restricting the robot's choices is acceptable (and allows the MDP to be solved more quickly by eliminating large parts of the search space).

In the following section we continue our discussion and numerical comparison of computing exact optimal policies by SPUDD with computing policies optimal with respect to constraints provided by a Golog program. Specifically, we are interested in comparison of *time* required for computing policies in both cases and *quality* of policies computed in both cases (in terms of values of the initial states delivered by policies). In general case, we can expect that policies constrained by Golog programs will not be exactly optimal, but we hope that they will be not too far from optimal policies and they will be computed faster than exact optimal policies.

5.5 A time and quality comparison of DTGolog with SPUDD

In this section, we consider a well known FACTORY example. This example received significant attention in [Hoey *et al.*, 1999, Hoey *et al.*, 2000], where it was used to compare computation times required by SPUDD with running times demonstrated by a *structural policy iteration* implementation [Boutilier *et al.*, 1995, Dearden and Boutilier, 1997]. We would like to use the same benchmark domain also because this domain has natural constraints on actions that can be easily expressed in a Golog program.[17]

A FACTORY problem can be easily expressed using the situation calculus based representation introduced in this chapter. There are six stochastic agent actions that can be applied to parts that require different types of processing. Five of them have only two outcomes (action succeeds or fails): $shape(x)$, $handPaint(x)$, $polish(x)$, $drill(x)$, $weld(x, y)$, and one action $sprayPaint(x)$ has four outcomes available for nature (they represent different quality of painting). In addition, there are three deterministic agent actions: $dip(x)$ (paint x by dipping it), $bolt(x, y)$, $glue(x, y)$ (connect parts x and y together by bolting them or by gluing them). To represent state features of this domain, it is sufficient to introduce 23 fluents. The most important are the following fluents.

- $connected(x, y, s)$, $connectedWell(x, y, s)$: parts x and y can be simply connected in s, e.g., by gluing, or well connected, e.g., by welding them. Well connected parts remain well connected unless the agent shapes one of them.

- $painted(x, s)$, $paintedWell(x, s)$: a part x can be painted well in s or not well. A painted part remains painted unless it is shaped, polished or drilled.

[17] Data for all six ADD-based FACTORY problems of increasing complexity are freely available for download from the Web page at: www.cs.ubc.ca/spider/staubin/Spudd/index.html

- $shaped(x, s)$, a part x can be shaped if the stochastic agent action $shape(x)$ succeeds. A shaped part remains shaped unless it is (or a part to which it is connected) is drilled or shaped.

- $smoothed(x, s)$, a shaped part x becomes smoothed if the stochastic agent action $polish(x)$ succeeds. A part remains to be smoothed in the next situation unless it is (or a part to which it is connected) is shaped or drilled.

- $drilled(x, s)$, a part x can be drilled if the agent has a drill and bits.

All other fluents are predicates with situation s as the only argument and their truth value remains the same in the successor situation $do(a, s)$ as it was in s, no matter what action the agent did. For example, there are fluents $skilledLabourPresent(s)$, $hasSprayGun(s)$, $hasGlue(s)$, $hasBolts(s)$, $hasDrill(s)$, $hasArcWelder(s)$, $hasSpotWelder(s)$, etc. All fluents are characterized by the appropriate successor state axioms, possibility of deterministic (the agent's or nature's actions) is specified by the precondition axioms and probabilities of nature choices are characterized by definitions. All logical descriptions follow ADD-based representations of FACTORY problems. Because in the domain of FACTORY problems there are either two parts or three parts (mentioned in ADD representations of *factory5.dat* and *factory6.dat* problems only), this can be reflected in the predicate logic axiomatization by using the domain closure axiom:

$$object(x) \equiv x = A \vee x = B \vee x = C$$

A Prolog implementation of the situation calculus representation of the *factory6.dat* (most complex) problem is attached in Appendix C.5. All other simpler problems are represented by a simplified subsets of this problem. There are minor differences in the two representations. More specifically, because the situation calculus representation is using action terms $weld(x, y)$ and $glue(x, y)$, where the variables x and y vary over objects $\{A, B, C\}$ allowed in the 6th factory domain, the logical representation allows several additional actions (e.g., $weld(A, C)$, $weld(B, C)$, $glue(A, C)$, $glue(B, C)$) that are not allowed and not considered in the *factory6.dat* example. A reward function in each Prolog implementation is modeled after the corresponding reward function in an ADD-based problem.

The following Golog program expresses natural constraints in this domain. It consists of the sequential composition of two different while-loops. The first while-loop makes sure that parts are shaped and connected. First, we check whether there is a non-shaped part, if yes, it gets shaped; otherwise, we pick up two parts that are not connected and either drill and bolt them or (if a skilled welder is present) weld them (if there is no skilled welder, we glue parts). The second while-loop commands to polish parts and paint them.

proc $processParts$
while ($\exists x, y.object(x) \wedge object(y) \wedge x \neq y \wedge \neg(shaped(x) \wedge connectedWell(x,y))$) **do**
  if $\exists z(object(z) \wedge \neg shaped(z))$
   then $\pi z.(\, object(z) \wedge \neg shaped(z))? \,;\, shape(z)\,)$
    else $\pi x.(\exists w.object(x) \wedge object(w) \wedge x \neq w \wedge \neg connectedWell(x,w))? \,;$
      $\pi y.(object(y) \wedge x \neq y \wedge \neg connectedWell(x,y))? \,;$
       $(\, drill(x) \,;\, drill(y) \,;\, bolt(x,y)$
        $|$
       if $hasSkilledWelder$ **then** $weld(x,y)$ **else** $glue(x,y)\,)$
      $)\,)$
endWhile ;
while ($\exists x.object(x) \wedge \neg paintedWell(x)$) **do**
  if $\exists z.(object(z) \wedge \neg smoothed(z))$
   then $\pi z.(object(z) \wedge \neg smoothed(z))? \,;\, polish(z))$
    else $\pi x.(object(x) \wedge \neg paintedWell(x))? \,;$
     if $skilledLabourPresent$ **then** $handPaint(x)$ **else** $(spray(x) \mid dip(x))$
     $)$
endWhile
endProc

Results, displayed in Table 5.1, are presented for $BestDo$ running on six FACTORY examples, and for SPUDD running on five. We could not complete computing an optimal policy using SPUDD for the largest *factory6.dat* example on our machine after running it continuously for 12 hours. However, [Hoey *et al.*, 2000] reports 3676 seconds running time for this example on Sun SPARC Ultra 60 running at 300Mhz with 1Gb of RAM (apparently, using a different implementation of SPUDD; we used the version 3.2.2). For all other examples, in comparison with running times reported in [Hoey *et al.*, 2000], running SPUDD on our machine takes less time (by the factor 2 or 3) which is not surprising giving that our machine has a faster CPU (and the same amount of RAM).

SPUDD was run using binary-valued variables only. Note that both SPUDD and $BestDo$ interpreter solve a finite horizon optimization problem. We tried each example with the value of horizon $H = 12$ (with an exception of the *factory6.dat* example, where $H = 15$, because this example involves more objects and primitive actions and requires a longer horizon to process

Ex.	State space size		SPUDD		BestDo		
	vars	states	time (sec)	value (S_0)	time (sec)	value (S_0)	succ. prob.
f	17	1.31×10^5	2	40.05	2	12.48	0.64
f0	19	5.24×10^5	3	40.05	2	12.48	0.64
f1	21	2.10×10^6	6	40.05	2	12.48	0.64
f2	22	4.19×10^6	7	40.05	3	12.48	0.64
f3	25	3.36×10^7	21	61.17	17	20.37	1.0
f4	28	2.68×10^8	49	61.17	19	20.37	1.0
f5	31	2.15×10^9	110	20.74	20	11.31	1.0
f6	35	3.44×10^{10}	–?–	–?–	304	22.98	1.0

Table 5.1: Results for FACTORY examples.

all objects). Running times (rounded to the nearest integer value) are shown both for SPUDD (version 3.2.2) and $BestDo$.[18] Note that implementation details of the SPUDD algorithm (value iteration with ADDs) and of the $BestDo$ interpreter, as well as different programming languages used in implementations (C++, in the case of SPUDD, and Prolog, in the case of $BestDo$) somewhat cloud comparisons of running times. Nevertheless, data in the table give certain indications about efficiencies of these two particular implementations. The table includes also values of the initial state relative to optimal policies computed by SPUDD and relative to policies computed by $BestDo$. The last column includes success probabilities for optimal policies computed by $BestDo$. These probabilities are 0.64 because in the examples f, f0, f1 and f2 in this domain the primitive action $weld(x, y)$ is not available and parts can be connected only by actions $glue(x, y)$ or $bolt(x, y)$. The latter assures that parts will be well connected and for this reason it is better than the former. Because the action $bolt(x, y)$ is possible only if both parts have been already drilled and the stochastic action $drill(z)$ succeeds only with the probability 0.8, the action $bolt(x, y)$ achieves a desired result only with the probability 0.64. (It might happen that the first while-loop, it makes sure that parts are connected, will be terminated at the decision epoch 12 before it achieves the intended effect.) In the case of examples, f3, f4, f5 and f6 the success probability is 1.0 because both outcomes of the stochastic action $weld(x, y)$ are always possible (if there is a skilled welder) without preliminary processing of parts x and y, and this action is included in the computed optimal

[18]For an infinite horizon optimization problem with the discount factor 0.9 and tolerance 1.8 the running times for SPUDD are about two times longer; it takes SPUDD 18 iterations to converge in this case.

policy.

In addition, it is interesting to compare data in our table with data in [Hoey *et al.*, 2000], where it is reported that flat (unstructured) value iteration took 895 and 4579 seconds when it was run on *factory.dat* and *factory0.dat* examples, respectively. These two examples have smallest state spaces; memory limitations precluded completion of the flat algorithm for more complex examples.

Our comparison demonstrates that there is a loss in terms of the value (of an initial state) delivered by an optimal policy computed by DTGolog in comparison to the value (of the same initial state) delivered by an optimal policy computed by SPUDD. This loss is what we expected because our Golog program limits the set of policies that can be compared and $BestDo$ finds a policy that is optimal with respect to the constraints provided by the program, but SPUDD computes an exact optimal policy (without any constraints). Indeed, as we have mentioned in Section 2.4.2, the value $V(S_0)$ of the root S_0 of a decision tree is equal to the value $V_n(S_0)$ computed using value iteration (2.25). In an unconstrained search tree, all actions possible in S_0 (and in subsequent reachable states) must be taken into account to compute the value of S_0. This computation would correspond to using $BestDo$ to compute a value of S_0 when a Golog program is simply a nondeterministic choice between all actions available in the application domain. Because our Golog program limits the number of actions that can be executed in S_0 and in subsequent situations, and, as a consequence, fewer states contribute to the value of each action (given by the expected value of its successor states when averaging operation is applied), the resulting value of $V(S_0)$ is less than it would be if all branches of the search tree with all leaves were taken into account (recall that values of leaves are determined by the reward function). However, as far as the computation of an optimal policy is concerned, this loss in the absolute value of $V(S_0)$ is less critical if actions that are optimal in the unconstrained decision tree remain optimal in the partial tree circumscribed according to the program constraints. Because an optimal action in each state is determined by the maximization operation (both in the equation (2.25) and in the semantics of $BestDo$), absolute values are less important than the fact that the same action yields the maximum values in both cases (the values themselves are different for reasons that we just explained). A detailed analysis of an optimal policy computed by SPUDD (using a provided graphical tool) and a policy computed by $BestDo$ reveals that the policies are identical on the subset of the state space that is reachable from S_0 (by using the actions that agree with constraints imposed by the Golog program). In other words, a loss in terms of value $V(S_0)$ does not have dramatic effect on an optimal policy itself. Table (5.1) demonstrates also that the difference between values in both cases is accompanied by a decrease in terms of computation time required to determine an optimal policy. Despite the drawbacks of an off-line decision tree search, this effect is achieved because the benchmark domain has natural constraints on actions. These natural constraints are easy to exploit in a simple Golog program to achieve performance comparable with performance of a state-of-the art MDP solver.

Our experimental results indicate that further effort in comparison of policies computed by $BestDo$ with policies computed by SPUDD would be useful. This effort should concentrate on those domains with many different stochastic actions which have natural constraints on the order when these actions can be executed (because the natural structure of these domains can be exploited in Golog programs). One possible example of an application domain of this type is the robots with multiple links such that motion of each link can be characterized by a stochastic action. Direct programming of these robots is difficult because they have a large number of degrees of freedom and choosing correct numerical values for each individual motion is an error prone and cumbersome process. Solving an MDP problem for these robots is usually impossible (or very difficult), because of the size of the state and action spaces. On the other hand, DTGolog can be useful, because the coordinated motion of all these links usually has a natural structure that can be easily programmed, but fine tuning of individual motions may require search in the space of parameters associated with each motion. In the next section, we describe our experience with robot programming for the case of a mobile robot that has a few degrees of freedom, but nevertheless, this robot solves an MDP problem with a very large state space.

5.6 Robot Programming: Our Experience and Advantages

In this section we discuss our implementation of DTGolog on a mobile robot B21 (see Section 2.5 for details). The robot navigates using BeeSoft [Burgard *et al.*, 1998, Thrun *et al.*, 1999], a software package that includes methods for map acquisition, localization, collision avoidance, and on-line path planning.

A key advantage of DTGolog as a framework for robot programming and planning is its ability to allow behavior to be specified at any convenient point along the programming/planning spectrum. By allowing the specification of stochastic domain models in a declarative language, DTGolog not only allows the programmer to specify programs naturally (using high-level robot actions as the base level primitives), but also permits the programmer to leave gaps in the program that will be filled in optimally by the robot itself. This functionality can greatly facilitate the development of complex robotic software. The planning ability allows for the scheduling of complex behaviors that are difficult to preprogram. It also obviates the need to reprogram a robot to adapt its behavior to reflect environmental changes or changes in objective functions. Programming, in contrast, is crucial in alleviating the computational burden of uninformed planning.

To illustrate these points, we have developed a program for our mobile delivery robot, tasked to carry mail and coffee in our office building. The task of DTGolog is to schedule the individual deliveries in the face of stochastic action effects arising from the fact that people may or may not be in their office at the time of delivery. It must also contend with different priorities for different people and balance these against the domain uncertainty.

Suppose that the robot's objective function is given by a reward function that associates an independent, additive reward with each person's successful delivery. The relative priority associated with different recipients is given by this function. For example, we might use *reward*$(Visitor, t, s) = 15 - t/20$ and *reward*$(Ray, t, s) = 30 - t/10$, where the initial reward (30) and rate of decrease (1/10) indicates relative priority.

Our robot is provided with the following simple DTGolog program:

while ($\exists p, n.\ \neg attempted(p) \wedge$ *mailPresent(p,n)*)
    $\pi(p\ :\ people)$
        $(\neg attempted(p) \wedge \exists n$ *mailPresent(p,n)* $)?\ ;\ deliverTo(p)$)
endWhile

Intuitively, this program chooses people from the finite range *people* for mail delivery and delivers mail in the order that maximizes expected utility (coffee delivery can be incorporated readily). The procedure *deliverTo* is a conditional sequence of actions similar to the procedures considered in Section 5.4. But this sequence is a very obvious one to hand-code in our domain, whereas the optimal ordering of delivery is not (and can change, as we'll see). We have included a guard condition $\neg attempted(p) \wedge \exists n$ *mailPresent(p,n)* in the program to prevent the robot from repeatedly trying to deliver mail to a person who is out of her office. This program constrains the robot to just one attempted mail delivery per person, and is a nice example of how the programmer can easily impose domain specific restrictions on the policies returned by a DTGolog program.

Several things emerged from the development of this code. First, the same program determines different policies—and very different qualitative behavior—when the model is changed or the reward function is changed. As a simple example, suppose that it takes 45 units of time to go from the main office to the visitor's office and it takes 110 units of time to travel from the main office to the office of Ray. If the probability that Ray (high priority) is in his office is 0.8, his delivery is scheduled before visitor's (low priority) if the visitor is in his office with the probability 0.6. The robot gets higher reward if it delivers mail first to Ray at 110, returns to the main office at 220 and delivers mail to the visitor at 265.[19] But when the probability that Ray (high priority) is in his office is lowered to 0.6, visitor's delivery is scheduled beforehand: the robot delivers mail first to the visitor at 45 and later to Ray at 200.[20] Such changes in the domain would require a change in the control program if not for the planning ability provided by DTGolog. The computational requirements of this decision making capability are much less than those should we allow completely arbitrary policies to be searched in the decision tree.

Full MDP planning can be implemented within DTGolog by running it with the program that allows *any* (feasible) action to be chosen at *any* time. This causes a full decision tree to be constructed. Given the domain complexity, this unconstrained search tree could only be completely evaluated for problems with a maximum horizon of seven (in about 1 minute)—this depth is barely enough to complete the construction of a policy to serve one person. With the program above, the interpreter finds optimal completions for a 3-person domain in about 1 second (producing a policy with success probability 0.94), a 4-person domain in about 9 seconds (success probability 0.93) and a 5-person domain in about 6 minutes (success probability 0.88). The latter corresponds to a horizon of about 30; clearly the decision tree search would be infeasible without the program constraints (with size well over 5^{30}). We note that the MDP formulation of this problem, with 5 people and 7 locations, would require more than 4 billion states (using the formula $32^P \cdot P^3$ from Section 5.4). So it would be difficult to use dynamic programming to solve this MDP without program constraints (or exploiting some other form of structure).

These experiments illustrate the benefits of integrating programming and planning for mobile robot programming. We conjecture that the advantage of our framework becomes even more evident as we scale up to more complex tasks. For example, consider a robot that serves dozens of people, while making decisions as to when to recharge its batteries. Mail and coffee requests might arrive sporadically at random points in time, not just once a day (as is the case for our current implementation). Even with today's best planners, the complexity of such tasks is well beyond what can be tackled in reasonable time. DTGolog is powerful enough

[19]The accumulated expected reward in this case is 16.1 assuming that the probability of the successful arrival is 0.99 and $H \geq 12$.

[20]The total expected reward for this optimal policy is 13.4 .

to accommodate such scenarios. If supplied with programs of the type described above, we expect DTGolog to make the (remaining) planning problem tractable—with minimal effort on the programmer's side.

5.7 Discussion

This chapter is partially based on our paper [Boutilier *et al.*, 2000a]. Our representation is related to the representations proposed in [Bacchus *et al.*, 1995, Bacchus *et al.*, 1999, Poole, 1997]. In particular, our decomposition of stochastic actions into deterministic nature's choices with associated probabilities inherits ideas formulated in [Bacchus *et al.*, 1999]. Our definition 5.2.1 of a policy is slightly different from that of (Definition 12.5.3) in [Reiter, 2001a]: we allow in *senseEffect*(a) any sequences of sensing actions, not just a single sensing action. In addition, our sense conditions are arbitrary situation suppressed sentences. Our implementation of the full observability assumption also has minor differences from [Reiter, 2001a] (see Definition 12.5.4, p. 369), where the agent uses one of the stochastic sense actions to find an outcome of the most recent stochastic physical action and it is assumed that sensing is accurate: the agent senses exactly the outcome of its stochastic action. In our case, we assume that sense actions included in *senseEffect*(a) procedure return accurate data at the run time, and because sense actions occur in the successor state axioms (see Chapter 3), the agent can find the actual outcome by evaluating one or several sense conditions. [Poole, 1997] embeds probability and decision theory into the situation calculus, but his ontology is substantially different from other proposals introducing stochastic actions into the situation calculus. In particular, the framework proposed by Poole does not decompose nondeterministic actions into a set of deterministic actions that nature can perform, rather his primitive actions are typically nondeterministic, and correspond to those actions that the agent can perform (but with nondeterministic outcomes chosen by nature). These nondeterministic actions are mentioned in the situation terms and they also occur in the successor state axioms; for this reason, writing axioms can be somewhat problematic.

[Grosskreutz, 2000, Grosskreutz and Lakemeyer, 2001] consider probabilistic extensions to Golog differently from our approach.

There are several proposals on using search to solve MDPs in cases when the state space is very large, e.g., [Barto *et al.*, 1995, Dearden and Boutilier, 1997, Koenig and Simmons, 1995], these papers use heuristic value function estimates and others. We reviewed the related literature in the end of Section 2.4.1. These proposals demonstrate that interleaving decision-theoretic planning with execution provides significant computational advantages in comparison to purely offline planning. However, $BestDo$ does offline planning only. In the next chapter, we consider another interpreter that combines offline planning with online execution.

Recent attempts to deal with the large state space have turned to principled ways of using temporally extended partial policies, where decisions are not required at each decision epoch, but rather invoke the execution of activities extended over several decision epochs and these activities follow their own policies until termination. This leads naturally to hierarchical control architectures and learning algorithms.

Since early days of AI, the idea of using more abstract actions to facilitate planning has been in a focus of AI research [Fikes *et al.*, 1972, Sacerdoti, 1974]. The main idea is to augment the primitive actions available to solve a given planning task with *macro-actions*, open-loop sequences of actions that can achieve some subgoal. A related issue is learning useful macro-operators, which can be subsequently re-used to solve different (but related) planning problems [Korf, 1985a, Korf, 1985b, Minton, 1988]. In particular, Korf introduced the notion of independent and serializable subgoals, which provide a decomposition of a planning problem. Each subgoal can be solved individually, and then the corresponding macro-operators can be combined by sequencing to solve the larger planning problem. Korf proposed a generate-and-test approach for constructing the macro-operators.

Inspired by research on hierarchical planning, [Singh, 1992a, Singh, 1992b] consider a class of sequential decision tasks, called composite sequential decision tasks, in which each of a sequence of subgoals must be achieved in turn. Compositionally structured tasks provide the possibility of sharing knowledge across the many tasks that have common subtasks and lead to a natural abstraction hierarchy consisting of macro operators. This is a special case of an MDP that may be decomposed into independent subproblems, but it is an interesting class because planning at higher levels of abstraction does not result in sub-optimal policies.

In a more general setting, [Dean and Lin, 1995a, Dean and Lin, 1995b] investigate methods that given a partition of the state space into several regions[21] can decompose a large decision-theoretic planning problem into a number of local problems, solve the local problems in each of these regions, and then combine the local solutions to generate a global solution. The authors present an algorithm for combining solutions to subproblems in order to generate an optimal solution to the global problem. This algorithm, called iterative approximation approach, relies on constructing an abstract MDP by considering individual regions as abstract states and their can-

[21] It is assumed that a domain expert provides a natural partition of the state space.

didate local policies as abstract actions. Building on the idea of Dantzig-Wolfe decomposition principle [Dantzig and Wolfe, 1960], the proposed algorithm iteratively discards the old local policy (that was considered optimal in a region on a previous iteration), and computes a new policy for each region based upon the abstract problem's estimates for the values of "important" states. As a result, each iteration leads to an improvement of a global abstract MDP. This algorithm is based on identification of an important set of *exit periphery states* for each region (these are states outside the region that can be reached in a single transition from states inside the region) and on identification of a set of *boundary states* for each region (these states are on the boundary of the region, from each of them the agent can move outside the region in a single transition). The states in the exit periphery of regions from the provided partition are important because they describe communication links between regions in the partition. Values of states in the exit periphery can be considered as a summary of interactions between regions. These values are instrumental to determine local policies (abstract actions) which in turn determine a global policy on the entire state space. The iterative algorithm successively modifies values of states from the exit periphery on each iteration and converges to an optimal global policy in a finite number of iterations. Ideas developed in [Dean and Lin, 1995a, Dean and Lin, 1995b] influenced several subsequent developments of methods designed to solve large MDPs using macro-actions [Hauskrecht *et al.*, 1998, Parr, 1998a, Parr, 1998b].

Building on the work of [Sutton, 1995], Sutton and his colleagues introduced models of *options* (also known as *macro-actions* and *abstract actions*) for MDPs [Precup and Sutton, 1998, Precup *et al.*, 1998, Sutton *et al.*, 1999, Precup, 2000]. Intuitively, an option is a closed-loop policy (for a particular region of the state space) that dictates what actions must be taken over a sequence of decision epochs.[22] This policy can be initiated when an agent enters this region, then the agent uses consecutively this policy to make decisions regarding which actions to take, and finally, this policy terminates when the agent leaves this region. When an option terminates, the agent selects another option to be executed. The term "options" (for courses of actions that can occupy only one or more than one decision epoch) is used to emphasize that primitive actions are a special case of one-step options and they are treated similarly with non-primitive actions. [Precup, 2000] considers two kinds of representations for options: a flat representation, in which an option chooses among primitive actions, and a hierarchical representation, in which an option chooses among other options. A very important idea of the option

[22]The set of primitive actions available to an option is always extended by an auxiliary *reset action* that represents the decision to terminate a policy.

framework is that an option can be treated as if it were a primitive action in the original MDP once an appropriate reward and transition models are defined for each option. Formally, a set of options defined over an MDP constitutes a discrete-time SMDP embedded within the original MDP. As a consequence, the options have internal structure (in terms of the underlying MDP) that can be exploited to change existing options and to learn new options. Using a grid-world domain, [Sutton *et al.*, 1999, Precup, 2000] demonstrate that proper options can significantly speed-up the search for optimal polices. However, in some cases, decision-theoretic planning with a given set of options can lead to adverse effects. These adverse effects can be alleviated by looking inside options and by altering their internal structure to better fit a planning problem at hands. In particular, using temporal-difference methods, Sutton and his colleagues develop *intra-option learning* methods to learn usefully about options (before they terminate) from fragments of experience within each of the options.

Developing the option framework, [Hauskrecht *et al.*, 1998] consider a problem of constructing an abstract MDP that consists of states on the borders of adjacent regions only, and its set of actions includes macro-actions only. In some application domains, this hierarchical MDP has much smaller state space than an initial MDP with options and, as a consequence, it can be solved more efficiently. In addition, the authors discuss how proper macro-actions can be generated to ensure approximately optimal performance. They also argue that additional computational burden associated with computing a set of good macro-actions can be compensated when macro-actions are subsequently re-used when solving multiple related MDPs. In a special restricted class of MDPs with "weakly coupled" regions (i.e., the number of states connecting any two regions is small), [Parr, 1998a] proposes more efficient algorithms (in comparison to the algorithm proposed in [Hauskrecht *et al.*, 1998]) for computing sets of useful macro-actions for each of the regions in the decomposition.

The $BestDo$ interpreter specified in this chapter, does not have any programming constructs related to decomposition of an MDP. However, these constructs can be easily introduced and the construct $local(\delta_1); \delta_2$ defined in the next chapter is motivated by the previous work mentioned above. Note also that in our approach Golog programs can be constructed from primitive actions only, and we do not consider macro-actions (options) as new building blocks for designing Golog programs. However, there are no conceptual limitations on introducing options in this framework. Indeed, there are several developments related to exploring macro-actions in the framework of the situation calculus with stochastic actions. For example, [Ferrein *et al.*, 2003] extends DTGolog framework with options and provides experimental evaluation using grid worlds. [Gu, 2002] explores whether macro-actions allow to speedup solution of the forward projection task when all actions are stochastic and reports experimental evaluation demonstrating that macro-actions provide computational savings.

The *hierarchies of abstract machines* (HAMs) approach is developed in [Parr and Russell, 1998, Parr, 1998b], where the authors consider specifying partial policies using stochastic finite-state machines and develop an algorithm to construct a policy that is optimal subject to these constraints. [Parr, 1998b] notes that "a cornerstone of the HAM approach is the idea that policies should be thought of as *programs* that produce actions as some function of the agent's sensor information" (p.89). HAMs implement this idea by allowing policies to be specified in terms of hierarchies of stochastic finite-state machines. Input sets of all machines are equal to the set of states $\mathcal{S}$ of the given finite MDP. Output sets of all machines are equal to the finite set of actions $\mathcal{A}_s$ available in the state $s \in \mathcal{S}$. For a machine $\mathcal{H}_k$ with a finite set of states $\mathcal{M}_k$, the transition function $\delta_k(m, x, m')$ is a probability defined on the cross-product of $\mathcal{M}_k$ and $\mathcal{S}$ as follows. If the current machine state is $m \in \mathcal{M}_k$ and the current state of the MDP is $x \in \mathcal{S}$, then the machine makes a transition to the state $m' \in \mathcal{M}_k$ with the probability $\delta_k(m, x, m')$. The states in each machine can be of five types: $\{$ *start, stop, action, call, choice* $\}$. In particular, each machine has a single distinguished *start* state and may have one or more distinguished *stop* states. Each machine $\mathcal{H}_k$ has an associated initialization function $\mathcal{I}_k : \mathcal{S} \times \mathcal{M}_k \to [0, 1]$ from environment states and machine states that determines the probability of the initial machine state that can be an action state, a choice state or a call state. Whenever $\mathcal{H}_k$ starts executing in the state $s \in \mathcal{S}$ of the underlying MDP, control starts at the state $m \in \mathcal{M}_k$ with the probability $\mathcal{I}_k(m, s)$; stop states return control back to the calling machine. The most interesting states are action states. An *action* state of the currently executing machine $\mathcal{H}_k$ generates an action $a \in \mathcal{A}_s$ based on the current state $s \in \mathcal{S}$ of the MDP and the current state $m \in \mathcal{M}_k$ according to the output function $\pi_k(m, s)$ of the machine $\mathcal{H}_k$. Upon receiving an action a, the MDP makes a stochastic transition to a next state and provides an immediate reward. A *call* state suspends execution of the currently executing machine $\mathcal{H}_k$ and calls another HAM, say $\mathcal{H}_j$, as a subroutine, where j is a function of the current state m of the machine $\mathcal{H}_k$. The HAM $\mathcal{H}_j$ starts execution in a state n with the probability determined by the initialization function $\mathcal{I}_j(n, s)$, where s is the current state of the controlled MDP. When $\mathcal{H}_j$ will reach one of its stop states, the control returns to $\mathcal{H}_k$ whose execution continues. [Parr, 1998b] requires that the call graph of each machine must be a tree. A *choice* state nondeterministically selects a next machine state from the finite set of available choices. Using choice states, a programmer can specify a controller for an MDP partially. It is assumed that the top-level machine in the hierarchy does not have a stop state and, thus, never terminates. It is assumed also there are no infinite (probability 1) loops that do not contain action states. A HAM $\mathcal{H}$ is defined by the top-level machine and the closure $\mathcal{S}_{\mathcal{H}}$ of all machine states of all machines reachable from the possible initial states of the initial machine. [Parr and Russell, 1998, Parr, 1998b] state and prove several important results about MDPs controlled by a HAM. First, it is proved that for any MDP and any HAM there exists an SMDP the solution of which defines an optimal choice function that maximizes the expected, discounted sum of rewards received by an agent executing the HAM. The state space of this SMDP is the cross-product of $\mathcal{S}$, the set of states of an MDP, and $\mathcal{S}_{\mathcal{H}}$, the set of reachable states in $\mathcal{H}$. All transitions in this SMDP are determined by transitions in $\mathcal{H}$ and in the controlled MDP. The only actions of this SMDP are choices in the choice states of $\mathcal{H}$. It is assumed that a HAM $\mathcal{H}$ generates typically primitive actions more than once between consecutive choice states to control the underlying MDP. All these primitive actions are determined by action states of $\mathcal{H}$ and they lead to rewards accumulated during these periods between choice states. Thus, parts of policies are constrained by the programmer who designs $\mathcal{H}$ using a domain specific knowledge about the controlled MDP. Second, [Parr, 1998b] proves that there exists a reduced SMDP, that is equivalent to a SMDP determined by a given pair of an MDP and a HAM, but this reduced SMDP contains only states determined by choice states of the HAM that controls the MDP. As a consequence, the procedural knowledge contained in a HAM is used to produce a reduced problem. The experimental results on solving an illustrative grid-world navigation problem with almost 3,700 states by using a simple HAM showed dramatic improvement in terms of the run time over the standard policy iteration algorithm applied to the original MDP without HAM; producing a good policy took less than a quarter of the time required to compute an optimal policy without HAM. A value of the policy constrained by the HAM was fairly close to a value of an actual optimal policy. Third, [Parr, 1998b] proves that a variation of Q-learning can be applied to this reduced SMDP with standard assumptions on parameters α that guarantee convergence to an optimal policy. Experiments with HAMQ-learning demonstrated even more dramatic speedup in comparison to traditional Q-learning: Q-learning required 9 million iterations to reach the level achieved by HAMQ-learning after 270,000 iterations.

Similar to the HAMs approach, the idea of finding an optimal policy consistent with constraints is also a key idea in the development of the DTGolog approach, but in the case of DTGolog constraints can be specified using standard (and less standard) programming constructs. The DTGolog framework has a number of advantages over HAMs. Finite-state machines are less convenient for programming than writing high-level programs in Golog. In addition, Golog programs can include while-loops and recursive procedures. Finally, primitive actions used in Golog programs are declaratively specified in Reiter's basic actions theories that provide a solution to the frame problem.

[Andre and Russell, 2001, Andre and Russell, 2002] provide further development of the HAM approach. In particular, [Andre and Russell, 2001] describes extensions to the HAM language that substantially increase its expressive power using parameterization (procedures can take a number of parameters the values of which must be filled in by the calling procedure), aborts and interrupts as well as memory variables (each of them has a finite domain). The authors provide a detailed example in the simulated Deliver-Patrol domain to argue that an extended language, called PHAM (programmable hierarchic abstract machines), allows to express the control program much more succinctly (using 9 machines) than HAM (the HAM program requires 63 machines). They mention also that the provided Deliver-Patrol program in PHAM induces less than 8,000 choice points in the joint SMDP, compared with more than 38,000 choice points in the original MDP. [Andre, 2003] proves that theoretical results about PHAMs mirror those obtained in [Parr and Russell, 1998, Parr, 1998b] about HAMs. [Andre and Russell, 2002, Andre, 2003] complement advantages provided by programming language features. Building on the state abstraction technique developed in [Dietterich, 2000], they propose a technique that groups certain sets of states together in equivalence classes (abstract states) if they have same values in some state features. They consider several types of equivalence conditions for states so that state abstractions can be applied safely, meaning that optimal policies in the abstract space are also optimal policies in the original space.[23] [Andre, 2003] proves that the abstracted Bellman equations for the Q-function describe the same problem as the non-decomposed Bellman equations, and that the optimal solutions to abstracted equations are the same as solutions to the non-decomposed Bellman equations. In addition to PHAM, [Andre and Russell, 2002, Andre, 2003] present ALisp, a Lisp-based high-level programming language with a nondeterministic construct that allows to specify partial controllers. This language subsumes earlier proposed languages such as MAXQ [Dietterich, 2000], options [Sutton *et al.*, 1999, Precup, 2000], and the PHAM language [Andre and Russell, 2001].

High-level programming languages with probabilistic statements have been in use for a long time. The most popular probabilistic statement in traditional programming languages is a random assignment operator. A formal analysis of probabilistic programs can be traced back to

[23] Optimality is understood as a *hierarchic optimality*, i.e., a policy is hierarchically optimal if it is optimal among all policies consistent with the given partial program.

[Kozen, 1981, Kozen, 1985, Feldman and Harel, 1984]. Other developments in this research area are summarized and briefly reviewed in [Harel *et al.*, 2000].

[Thrun, 2000] describes a programming language CES (an extension of C++) targeted towards development of robot control software. CES consists of C++ augmented with two ideas: Computing with probability distributions, and built-in mechanisms for learning from examples as a new means of programming. DTGolog is motivated by a similar intention of facilitating the development of robotics software, but it is based on a higher level of abstraction: primitive actions in DTGolog programs are characterized in the predicate logic using Reiter's basic action theories.

[McAllester, 2000] gives a model-checking style algorithm for verifying that a given handwritten controller achieves a desired value of decision-theoretic utility. It is asumed that both the controller and the model of an environment are written in a special purpose programming language described in the paper.

[Pfeffer, 2001] introduces a new programming language *IBAL* for probabilistic and decision-theoretic agents (IBAL stands for Integrated Bayesian Agent Language). *IBAL* integrates probabilistic reasoning, Bayesian parameter estimation and decision-theoretic utility maximization into the unified framework.

5.8 Future work

A number of interesting directions remain to be explored. The decision-tree algorithm used by the DTGolog interpreter is clearly subject to computational limitations.[24] However, the basic intuitions and foundations of DTGolog are not wedded to this particular computational model. We are planning to integrate efficient algorithms and other techniques for solving MDPs into this framework (dynamic programming, abstraction, sampling, etc.). We emphasize that even with these methods, the ability to naturally constrain the search for good policies with explicit programs is crucial. Other possible future avenues include: incorporating realistic models of partial observability (a key to ensuring wider applicability of the model); extending the expressive power of the language to include other extensions already defined for the classical Golog model (e.g., concurrency); incorporating declaratively-specified heuristic and search control information. In addition, it would be important to compare DTGolog with ALisp.

[24]Note, however, that program constraints often make otherwise intractable MDPs reasonably easy to solve using search methods.

Chapter 6

An On-line Decision-Theoretic Golog Interpreter.

6.1 Introduction

The aim of this chapter is to provide an on-line architecture for designing controllers for autonomous agents. Specifically, we explore controllers for mobile robots programmed in DTGolog, an extension of the high-level programming language Golog. DTGolog aims to provide a seamless integration of a decision-theoretic planner based on Markov decision processes with an expressive set of program control structures available in Golog. The DTGolog interpreter described in the previous chapter is an off-line interpreter that computes the optimal conditional policy π, the probability pr that π can be executed successfully and the expected utility u of the policy. The semantics of DTGolog is defined by the predicate $BestDo(\delta, s, h, \pi, u, pr)$, where s is a starting situation and h is a given finite horizon. The policy π returned by the off-line interpreter is structured as a Golog program consisting of the sequential composition of agent actions, *senseEffect*(a) sensing actions (which serve to identify a real outcome of a stochastic action a) and conditionals (**if** ϕ **then** π_1 **else** π_2), where ϕ is a situation calculus formula that provides a test condition to decide which branch of a policy the agent should take given the result of sensing. The interpreter looks ahead up to the last choice point in each branch of the program (say, between alternative primitive actions), makes the choice and then proceeds backwards recursively, deciding at each choice point what branch is optimal. It is assumed that once the off-line DTGolog interpreter has computed an optimal policy, the policy is given to the robotics software to control a real mobile robot.

However this off-line architecture has a number of limitations. Imagine that we are interested in executing a program $\delta = (\delta_1; \delta_2)$ where both sub-programs δ_1 and δ_2 are very large nondeterministic DTGolog programs designed to solve 'independent' decision-theoretic problems: δ_2 is supposed to start in a state belonging to a certain set, but from the perspective of δ_2 it is not important what policy will be used to reach one of those states. For example, imagine a cleaning robot that is charged with the task of computing and executing of an optimal policy of cleaning the first floor of a house and once this task is completed it has to move upstairs and clean the second floor. Intuitively, we are interested in computing first an optimal policy π_1 (that corresponds to δ_1), executing π_1 in the real world and then taking the remaining program δ_2, computing and executing an optimal policy π_2 from any of the states that will be reached by π_1. But the off-line interpreter can return only the optimal policy π_δ that corresponds to the whole program $(\delta_1; \delta_2)$, spending on this computation more time than necessary: decisions that have to be made during the execution of π_1 are not relevant to the task of computing and executing π_2. Indeed, the very structure of the $BestDo$ interpreter is based on the idea of considering all nondeterministic choices in δ_1 and looking ahead to the very last nondeterministic choice in $(\delta_1; \delta_2)$ before the policy π_δ can be computed. This requires much larger search space to be considered, but *it would be more practical to consider a subspace related to the sub-program* δ_1 *and save computation time by computing* π_1 *first.*

The second limitation becomes evident if in addition to *senseEffect* sensing actions serving to identify outcomes of stochastic actions, we need to explicitly include sensing actions in the program. Imagine that we are given a program

$$p_1; (\pi x, t).[(now(t))?; sense(Q, x, t); \textbf{if } \phi \textbf{ then } p_2 \textbf{ else } p_3],$$

where $sense(Q, x, t)$ is a sensing action that returns at time t a measurement x of a quantity Q (e.g., the current coordinates, the battery voltage) and the condition ϕ depends on the real data x that will be returned by sensors (in the program above p_1, p_2, p_3 are sub-programs, the nondeterministic choice operator[1] π binds variables x and t and the test $now(t)$? grounds the current time). Imagine also that sub-programs p_2 and p_3 are both very large programs. Intuitively, we want to compute an optimal policy (corresponding to p_1) off-line, execute this policy in the real world, sense v, find whether the expression $\phi(s)$ evaluates to true or to false in the situation that results from doing this sense action and, finally, choose the corresponding sub-program (either p_2 or p_3, exclusively) and compute and execute an optimal policy with respect to this Golog sub-program only without considering the second sub-program that corresponds

[1] It should be always clear from the context when we use π to denote a policy, and when we use π to denote a nondeterministic operator in a Golog program.

to an alternative truth value of $\phi(s)$. But the off-line interpreter is not able to compute an optimal policy with respect to a given program without considering both large sub-programs p_2 and p_3 in the case that the range of sensing values includes values that can lead to two different evaluations of $\phi(s)$.

We would like to develop an alternative interpreter that allows agents to save time on computing policies in circumstances outlined above. As our examples indicate, this alternative interpreter should be an on-line interpreter rather than an off-line interpreter because if a policy computed from an initial segment of a given program can be executed in the real world, then this execution leads to a specific state and a policy optimal with respect to a remaining part of the program can be computed from this specific state alone without taking into account all other alternative states that can be potentially reached by the initial policy. Because a policy is computed off-line from a given Golog program, an on-line interpreter must employ an off-line interpreter. At a first glance, we could simply employ the $BestDo$ interpreter and introduce new programming operators that a programmer can use to subdivide a whole Golog program into relatively independent parts. However, we would like to propose another off-line interpreter that provides same functionality as $BestDo$, by computing a policy π from a given Golog program δ, situation s and horizon h, and in addition computes from the program δ its sub-program γ that remains to be executed after actually performing the first action from the policy π. This new interpreter is called an *incremental off-line interpreter* and its semantics is defined in the next section using the predicate $IncrBestDo$. There are several reasons why this incremental off-line interpreter can be more useful than the previously introduced $BestDo$.

First, in many realistic scenarios only an incomplete, partial model is readily available to a Golog programmer. In particular, probabilities of all or some transitions between states can be unknown and can be learned only from ongoing interaction with an environment. In this case, the decision making agent can compute policies given a current estimate of the model, but once the agent executes actions online it must improve its model by learning about outcomes of stochastic actions, then compute a new policy optimal with respect to improved model, etc. More specifically, a partial model of stochastic transitions between states in an environment can account for none of the exogenous actions. Then, a policy computed by an off-line interpreter is optimal only relative to this approximate model of the world assuming there are no exogenous actions.[2] However, if the environment is not static, i.e., important exogenous actions do happen

[2]In other words, due to the lack of statistical data about occurrences of exogenous actions, a programmer can start with an approximation based either on the assumption that an environment is 'static', i.e., only the agent's actions can cause a transition from one state in the state space to another, or on the assumption that

and must be taken into account when the decision maker computes a policy, then to account for them at run time, the on-line interpreter can sense for them at each step when a first action from the policy is executed.[3] If an exogenous action really does occur, then it is possible that the policy that was provisionally considered optimal relative to the current model of the world (before the action occurred) will no longer be optimal. Then, the off-line interpreter has to re-compute a new policy optimal with respect to an expanded model. As a consequence, in a general case, the off-line interpreter does not need the remaining part of the policy π besides the very first action in this policy, but has to keep a 'remaining program' to continue the execution.

Second, there are stochastic actions that can lead to a large number of different states and because $BestDo$ does decision tree search, this search becomes time consuming if a branching factor is high, but if a stochastic action is executed on-line, then only one nature choice that does occur in reality matters. In the sequel, we use the phrase *a quasi-optimal policy* and the phrase an optimal policy instead of 'a policy optimal with respect to a given program and the current model of the world'.

In this chapter we introduce a new off-line decision-theoretic interpreter, *IncrBestDo*. In comparison with $BestDo$, it has one additional argument which is a remaining program.[4] Then, we propose to compute and execute policies on-line using this new incremental off-line decision theoretic interpreter. More specifically, we define an online interpreter that works in action-by-action mode as follows. Given a Golog program δ and a starting situation s, it computes a quasi-optimal policy π and the program γ, a sub-program of δ, that remains when a first action a in π will be executed. At the next stage, the action a is executed in the real world. The on-line interpreter gets sensory information to identify which outcome of a has actually occurred if a is a stochastic action: this may require doing a sequence of sensing actions on-line. The action a (and sensing actions performed after that) result in a new situation. Then the cycle of computing a quasi-optimal policy and remaining program, executing the first action and getting sensory information (if necessary) repeats until the program terminates or execu-

occurrences of exogenous stochastic actions are not 'folded in' a domain model of the agent's stochastic actions. Both assumptions can be used only temporarily and they will not be needed as soon as sufficient amount of data about exogenous actions will be collected by the agent.

[3]If a programmer knows that certain exogenous actions occur with a high probability at certain times – mail arrives in the main office every day at 10am – then the programmer can write a Golog program that commands to do appropriate actions. If probabilities of exogenous actions are not known in advance, then they can be accounted for at run time only.

[4]This remaining program can be loosely understood as a given program δ 'minus' the first action in a quasi-optimal policy. Recall that in Section 2.3.2, when we introduced *Trans*$(\delta, s_1, \gamma, s_2)$–based semantics for Golog we also considered a given program δ and a remaining program γ.

tion fails. In the context of this incremental on-line execution, we define a new programming operator $limit(\delta)$ that limits the scope of search performed by the off-line interpreter. If δ is the whole program, then no computational efforts are saved when our new off-line interpreter computes a quasi-optimal policy from δ: in this case, the operator $limit()$ makes no difference and can be simply ignored. But, if the programmer writes $limit(\delta_1); \delta_2$, then the on-line interpreter will compute (using the new incremental off-line interpreter) and execute action-by-action the Golog program δ_1 *without looking ahead to decisions that need to be made in* δ_2. Only when δ_1 has been executed to completion, in other words, only when all actions from a quasi-optimal policy constrained by δ_1 have been executed in the real world, only at this moment, will the on-line interpreter start to consider δ_2 for the first time. This can provide significant computational savings if δ_1 is a large program that includes many stochastic actions with a high branching factor because they result in a large number of alternative states, but if they are executed in reality, then only one of these states matters when the off-line interpreter starts to compute a policy from δ_2. If δ_2 is a large highly nondeterministic program, then it can be computationally infeasible to compute an optimal policy from $\delta_1; \delta_2$ without limiting the scope of search.

To provide an additional illustration of decision-theoretic subproblems constrained by δ_2 and δ_1 that are weakly coupled or independent, we continue with the sensing example mentioned above. If the programmer knows that the sensing action $sense(Q, x, t)$ is necessary to evaluate the condition ϕ, then using the program

$$limit(p_1)\ ;\ (\pi t, x).[(now(t))?\ ;\ limit(sense(Q, x, t))\ ;\ \ \mathbf{if}\ \phi\ \mathbf{then}\ p_2\ \mathbf{else}\ p_3]$$

the required information about Q will be obtained before the incremental interpreter will proceed to the execution of the conditional; therefore, only one of the two large programs p_3 or p_2 will be considered. The following simple scenario provides an illustration. Imagine an agent that has an airline ticket from Toronto to San Francisco and wants to compute an optimal policy for traveling from Toronto to a conference venue in Stanford. This computational task consists of several independent decision theoretic tasks: compute an optimal policy of traveling from home in Toronto to a local international airport and then execute it, sense the gate from where the flight will depart, take a plane and fly to an international airport in San Francisco and after arrival compute an optimal policy for traveling from the airport to the conference venue in Stanford. A Golog programmer does not need any information about the number of gates in Toronto and about assignment of flights to gates to write a program providing suitable constraints on the search for a globally optimal policy, given that the programmer uses the operator $limit(sense(Flight123Gate, x, t))$. In addition, using this operator, the programmer indicates that once the agent senses the right gate out of 100 gates available in the international airport in Toronto, and takes the flight, only after that the agent can start solving the decision theoretic planning task of traveling to Stanford.

Our discussions above suggest that the incremental interpretation of decision-theoretic Golog programs needs an account of sensing (formulated in the situation calculus). We propose to use the representation of sense actions introduced in Chapter 3. In Section 5.3, we demonstrated that this account is naturally applicable in the robotics context.

The remaining part of this chapter is structured as follows. In Section 6.2 we consider an incremental off-line decision-theoretic interpreter and in Section 6.3, an implementation of an online interpreter. Then, we describe the implementation on the B21 mobile robot. Section 6.5 discusses connections with related work.

6.2 The incremental off-line DTGolog interpreter

An on-line interpreter formulated in the next section uses an incremental off-line DTGolog interpreter, defined by the predicate *IncrBestDo*$(\delta_1, s, h, \delta_2, \pi, v, pr)$ as explained below. This predicate takes as input the Golog program δ_1, starting situation s, horizon h and returns a quasi-optimal conditional policy π, its total expected value v, the probability of success pr, and the program δ_2 that remains to be executed after doing the first action from the policy π. Note that in comparison to *BestDo*, *IncrBestDo* has an additional argument δ_2 representing the program that remains to be executed. The structure of *IncrBestDo* resembles the structure of *BestDo*. Given a Golog program δ_1, the interpreter looks ahead to the very last nondeterministic choice in every branch of δ_1, selects an optimal choice, proceeds backwards recursively to choose an optimal branch in the very first nondeterministic choice in δ_1 and keeps also the sub-program δ_2 that is the program mentioned in this first optimal choice in δ_1.

IncrBestDo$(\delta_1, s, h, \delta_2, \pi, u, pr)$ is defined inductively on the structure of a Golog program δ_1. All cases are defined similarly to *BestDo*, and we give them below with additional comments when necessary to explain differences between the two off-line interpreters.

1. **Zero horizon.**
   IncrBestDo$(\delta_1, s, 0, \delta_2, \pi, v, pr) \stackrel{def}{=} \pi = Nil \wedge \delta_2 = Nil \wedge v = reward(s) \wedge pr = 1.$
   If the horizon is zero, then we no longer need constraints in the program δ_1 and for this reason we can set the remaining program δ_2 to *Nil*.

2. **The null program**

$$\textit{IncrBestDo}(Nil, s, h, \delta_r, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\ \pi = Nil \wedge v = reward(s) \wedge pr = 1 \wedge \delta_r = Nil.$$

3. **The program δ_1 begins with a deterministic agent action.**

$$\begin{array}{l} \textit{IncrBestDo}(a; \delta_1, s, h, \delta_2, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\ \quad [\neg Poss(a, s) \wedge \\ \qquad \delta_2 = Nil \wedge \pi = \textit{Stop} \wedge pr = 0 \wedge v = \textit{reward}(s) \vee \\ \quad Poss(a, s) \wedge \delta_2 = \delta_1 \wedge \\ \qquad \exists(\gamma, \pi', v')\, \textit{IncrBestDo}(\delta_2, do(a, s), \gamma, h{-}1, \pi', v', pr) \wedge \\ \qquad\quad \pi = (a; \pi') \wedge v = reward(s) + v'\,]. \end{array} \tag{6.1}$$

If a deterministic agent action a is possible in situation s, then we compute the optimal policy π' of the remaining program δ_1, its expected utility v' and the probability of successful termination pr. Because a is a deterministic action, the probability that the policy π' will complete successfully is the same as for the program itself; the accumulated expected value v is a sum of a reward for being in s and the expected utility of continuation v'. If a is not possible, then the remaining program is Nil, and the policy is the *Stop* action, the expected utility is the reward in s and the probability of success is 0.

4. **First program action is stochastic.**
When a is a stochastic action for which nature selects one of the actions in $\{n_1, \ldots, n_k\}$, there are two cases depending on the policy that is computed using the auxiliary predicate *IncrBestDoAux* that takes all available nature's choices, the given program, the current situation, the number of stages to go and returns a quasi-optimal policy, its total expected value and the probability of success. (The definition of *IncrBestDoAux* is very similar to the definition of $BestDoAux$ given in the previous chapter.) First, when none of nature's actions are possible, as determined by *IncrBestDoAux*, or if only one of them is possible, but subsequent execution fails, then it returns the policy *Stop*, the program δ cannot be continued and the remaining program δ_r is *Nil* meaning that no constraints need to be considered any further. Second, when *IncrBestDoAux* finds that at least one of nature's actions is possible, then it returns the policy different from *Stop*, and the remaining program δ_r is simply δ.

$$\begin{array}{l} \textit{IncrBestDo}(a; \delta, s, \delta_r, h, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\ \quad \exists(\pi', v', x).\textit{IncrBestDoAux}(\textit{choice}'(a), a, \delta, s, h-1, \pi', v', pr) \wedge \\ \qquad \pi = a\,;\, \textit{senseEffect}(a)\,;\, \pi' \wedge v = reward(s) + v' \wedge \\ \qquad\quad (\pi' = (x)?; Stop \wedge \delta_r = Nil \vee \pi' \neq (x)?; Stop \wedge \delta_r = \delta\,) \end{array}$$

For *IncrBestDoAux*, we consider two cases: first, when $k=1$ and then $k \geq 2$. If $k=1$, then

$$
\begin{array}{l}
\mathit{IncrBestDoAux}(\{n_k\}, a, \delta, s, h, \pi, v, pr) \stackrel{def}{=} \\
\quad \neg \mathit{Poss}(n_k, s) \wedge \mathit{senseCond}(n_k, \phi_k) \wedge \pi = (\phi_k)?; \mathit{Stop} \wedge v = 0 \wedge pr = 0 \vee \\
\quad \mathit{Poss}(n_k, s) \wedge \mathit{senseCond}(n_k, \phi_k) \wedge \\
\qquad \exists(\gamma, \pi', v', pr')\, \mathit{IncrBestDo}(\delta, do(n_k, s), h, \gamma, \pi', v', pr') \wedge \\
\qquad\qquad \pi = (\phi_k)?\,;\, \pi' \ \wedge\ v = v' \cdot prob(n_k, a, s) \ \wedge\ pr = pr' \cdot prob(n_k, a, s).
\end{array}
$$

If $k \geq 2$, then

$$
\begin{array}{l}
\mathit{IncrBestDoAux}(\{n_1, \ldots, n_k\}, a, \delta, s, h, \pi, v, pr) \stackrel{def}{=} \\
\quad \neg \mathit{Poss}(n_1, s) \wedge \mathit{IncrBestDoAux}(\{n_2, \ldots, n_k\}, a, \delta, s, h, \pi, v, pr) \vee \\
\quad \mathit{Poss}(n_1, s) \ \wedge\ \exists(\pi', v', pr').\, \mathit{IncrBestDoAux}(\{n_2, \ldots, n_k\}, a, \delta, s, h, \pi', v', pr') \wedge \\
\qquad \exists(\gamma, \pi_1, v_1, pr_1).\, \mathit{IncrBestDo}(\delta, do(n_1, s), h, \gamma, \pi_1, v_1, pr_1) \ \wedge\ \mathit{senseCond}(n_1, \phi_1) \wedge \\
\qquad\qquad \pi = \textbf{if } \phi_1 \textbf{ then } \pi_1 \textbf{ else } \pi' \ \wedge \\
\qquad\qquad\qquad v = v_1 \cdot prob(n_1, a, s) + v' \ \wedge \\
\qquad\qquad\qquad\qquad pr = pr_1 \cdot prob(n_1, a, s) + pr'.
\end{array}
$$

Note that in the two above definitions and in (6.1) we do not use the program γ that remains to be executed after doing the first action in δ. The reason is that the interpreter computes γ only when the program starts with a stochastic or deterministic action a and in both cases we already know the remaining program: it is the program that follows a. We do not use γ because, as we will see in the section 6.3, the incremental off-line interpreter will be used by an on-line interpreter to compute a policy that is optimal relative to the current model of the world (in a general case, this model can be updated after doing a in the real world).

5. **The program starts with a test.**[5]

$$
\begin{array}{l}
\mathit{IncrBestDo}(\phi?; \delta_1, s, h, \delta_2, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\
\qquad \phi[s] \wedge \mathit{IncrBestDo}(\delta_1, s, h, \delta_2, \pi, v, pr) \vee \\
\qquad\qquad \neg\phi[s] \ \wedge\ \pi = Stop \ \wedge\ \delta_2 = Nil \ \wedge\ pr = 0 \ \wedge\ v = reward(s).
\end{array}
$$

6. **The nondeterministic choice of two programs.**

[5]Note that in the definition of the transition-based semantics for Golog in Section 2.3.2, evaluation of a test expression is considered as a single transition, but here evaluations of tests don't count as a transition.

$$
\begin{aligned}
&\mathit{IncrBestDo}((\delta_1 \mid \delta_2); \gamma, s, h, \delta_r, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\
&\quad \exists(\pi_1, v_1, pr_1, \delta^1_{rem}).\, \mathit{IncrBestDo}(\delta_1; \gamma, s, h, \delta^1_{rem}, \pi_1, v_1, pr_1) \wedge \\
&\quad \exists(\pi_2, v_2, pr_2, \delta^2_{rem}).\, \mathit{IncrBestDo}(\delta_2; \gamma, s, h, \delta^2_{rem}, \pi_2, v_2, pr_2) \wedge \\
&\qquad\qquad (\ (pr_1, v_1) \leq (pr_2, v_2) \wedge \pi = \pi_2 \wedge \ \delta_r = \delta^2_{rem} \ \wedge v = v_2 \wedge pr = pr_2 \ \vee \\
&\qquad\qquad\qquad (pr_1, v_1) > (pr_2, v_2) \wedge \pi = \pi_1 \wedge \ \delta_r = \delta^1_{rem} \ \wedge v = v_1 \wedge pr = pr_1 \ ).
\end{aligned}
$$

If the program starts with a nondeterministic choice between δ_1 and δ_2, then we look ahead up to the end of each of the two branches, and find which branch is optimal. If the first branch is optimal, then the remaining program δ_r is the remaining program δ^1_{rem} that was computed in the first branch; otherwise, if the second branch is optimal, then the remaining program δ_r is the remaining program computed in the second branch δ^2_{rem}.

7. **Conditionals.**

$$
\begin{aligned}
&\mathit{IncrBestDo}((\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2); \gamma, s, h, \delta_{rem}, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\
&\qquad\qquad \phi[s] \ \wedge \ \mathit{IncrBestDo}(\delta_1; \gamma, s, h, \delta_{rem}, \pi, v, pr) \vee \\
&\qquad\qquad\qquad \neg\phi[s] \ \wedge \ \mathit{IncrBestDo}(\delta_2; \gamma, s, h, \delta_{rem}, \pi, v, pr).
\end{aligned}
$$

8. **Nondeterministic finite choice of action arguments.**

If the program begins with the finite nondeterministic choice $(\pi(x : \tau)\delta); \gamma$, where τ is the finite set $\{c_1, \ldots, c_n\}$ and the choice binds all free occurrences of x in δ to one of these elements, then we have:

$$
\begin{aligned}
&\mathit{IncrBestDo}((\pi(x : \tau)\delta); \gamma, s, \delta_{rem}, h, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\
&\qquad \mathit{IncrBestDo}((\delta|^x_{c1} \ \mid \ \ldots \ \mid \ \delta|^x_{cn}); \gamma, s, \delta_{rem}, h, \pi, v, pr)
\end{aligned}
$$

where $\delta|^x_c$ means substitution of c for all free occurrences of x in δ. Thus, the optimal policy π corresponds to the element c in τ that delivers the best execution. Note that the remaining program δ_{rem} is the same on the both sides of the definition.

9. **Associate sequential composition to the right.**

$$
\begin{aligned}
&\mathit{IncrBestDo}((\delta_1; \delta_2); \delta_3, s, h, \delta_{rem}, \pi, v, pr) \stackrel{def}{=} h > 0 \wedge \\
&\qquad \mathit{IncrBestDo}(\delta_1; (\delta_2; \delta_3), s, h, \delta_{rem}, \pi, v, pr).
\end{aligned}
$$

10. **While-loop is the first program action.**

This specification requires second order logic; an implementation is provided in Appendix D.1.

11. **Procedure is the first program action.**

> There is also a suitable expansion rule when the first program action is a procedure call. This is similar to the rule for Golog procedures [Levesque *et al.*, 1997, De Giacomo *et al.*, 2000], and requires second order logic to characterize the standard fixed point definition of recursive procedures. An implementation is provided in Appendix D.1.

Recall that according to Definition 5.2.1, policies are Golog programs as well. Moreover, if p is a Golog program that contains many nondeterministic choices, a quasi-optimal policy π computed from p is a conditional program that does not involve any nondeterministic choices. This observation suggests that programmers may wish to take advantage of the structure in a decision theoretic problem and use explicit search control operators that bound the search for an optimal policy. As we discussed in an introduction to this chapter, in many realistic scenarios, full search in the decision tree is neither feasible nor desirable even if this search is constrained by a given program $(\delta_1; \delta_2)$. Instead of doing full search, an off-line interpreter (given a program δ_1) can begin with computing an optimal policy π_1 corresponding to a smaller local sub-space of the state space. Then, this policy can be expanded to a larger portion of the state space by computing a policy π optimal with respect to a large program $(\pi_1; \delta_2)$. Both these successive computations can be accomplished solely off-line. Alternatively, we can imagine, that a policy optimal with respect to δ_1, and, consecutively, policies, optimal with respect to remaining sub-programs of δ_1, are repeatedly computed off-line without consulting δ_2, but each off-line computation is followed by on-line execution of the first action from consecutively shorter policies. Once these on-line executions are completed in a particular state in the state space, search continues using another given program δ_2.

To implement these bounds on search, we introduce two new operators: $local(p)$ and $limit(p)$, in addition to the standard set of Golog programming constructs. The former corresponds to bounds on search done purely off-line, the latter corresponds to bounds on off-line search that is augmented with on-line executions. Intuitively, the program $local(p_1); p_2$ means the following. First, compute the optimal policy π_1 corresponding to the sub-program p_1, then compute the optimal policy π corresponding to the program $(\pi_1; p_2)$. If both sub-programs p_1 and p_2 are highly nondeterministic, then using the operator $local(p_1)$ the programmer indicates where the computational efforts can be saved: there is no need in looking ahead further than p_1 to compute π_1. As an example, consider $local(\gamma_1 \mid \gamma_2); \delta$. In this program, the programmer means that an off-line interpreter must find an optimal policy π_γ that is constrained either by the sub-program γ_1 or by the sub-program γ_2 without taking into account δ, but once π_γ has been computed, a policy optimal with respect to $local(\gamma_1 \mid \gamma_2); \delta$ can be computed off-line from the program $\pi_\gamma; \delta$. Formally, when a Golog program begins with $local(\delta_1)$ we have:

$$\begin{aligned} \textit{IncrBestDo}(local(\delta_1); \delta_2, s, h, \delta_{rem}, \pi, u, pr) \stackrel{def}{=} h > 0 \wedge \\ (\exists \gamma, \pi_1, u_1, pr_1) \textit{IncrBestDo}(\delta_1; Nil, s, h, \gamma, \pi_1, u_1, pr_1) \wedge \\ \textit{IncrBestDo}(\pi_1; \delta_2, s, h, \delta_{rem}, \pi, u, pr). \end{aligned} \tag{6.2}$$

In this definition, we compute a program γ, a sub-program of δ_1, that remains to be executed after actually performing the first action from π_1, but it cannot be used in the next computation to determine δ_{rem} because δ_1 can include several nested nondeterministic choices and the construct $local(\delta_1); \delta_2$ indicates that all nondeterministic choices in δ_1 must be resolved before considering δ_2. Therefore, the program δ_{rem} must be computed from the sequential composition $(\pi_1; \delta_2)$ in which π_1 is a policy (i.e., it is a conditional Golog program without nondeterministic choices but possibly with occurrences of stochastic actions); in addition, this sequential composition takes into account δ_2 as well.

Note that according to this definition, δ_{rem}, the program that remains to be executed, is computed off-line from a program that does not start with the $local$ operator, because by the definition of the policy, π_1 has no occurrences of $local$. Therefore, δ_{rem} can have occurrences of $local$ only if δ_2 has occurrences of $local$. Consequently, when a Golog program is $local(\delta_1); \delta_2$ the operator $local(\delta_1)$ can provide computational savings for an off-line interpreter only once by saying that δ_2 need not be considered to compute π_1. In other words, the construct $local(\delta_1)$ provides bounds on the search performed by an off-line interpreter, but once the policy π_1 was computed and the first action in π_1 was executed on-line, the remaining part of the program has no indication where the search efforts can be saved, unless δ_2 has other occurrences of search control operators. For this reason, the programmer may find it convenient to use another search control operator that once used persists in the remaining part of the program until the program inside the scopes of that operator terminates on-line. This operator is called $limit(p)$ and is specified by the following abbreviation.

$$\begin{aligned} \textit{IncrBestDo}(limit(\delta_1); \delta_2, s, h, \delta_{rem}, \pi, u, pr) \stackrel{def}{=} h > 0 \wedge \\ (\exists p')\, \textit{IncrBestDo}(\delta_1; Nil, s, h, p', \pi, u, pr) \wedge \\ (\, p' \neq Nil \;\wedge\; \delta_{rem} = (limit(p'); \delta_2) \vee \\ p' = Nil \;\wedge\; \delta_{rem} = \delta_2\,). \end{aligned} \tag{6.3}$$

According to this specification, an optimal policy π can be computed without looking ahead to the program δ_2; hence, using $limit(\delta_1)$ a programmer can express a domain specific procedural knowledge to save computational efforts. Informally, $limit(\delta_1); \delta_2$ is saying that an optimal policy can be determined solely with respect to δ_1 and nothing in δ_2 matters as long as a policy optimal with respect to δ_1 contains more than one action. Note that when p' is *Nil*, i.e. there will be nothing in δ_1 to execute after doing the only action in π, the remaining program δ_{rem} contains only δ_2. In addition, note that in comparison with $local$, δ_{rem}, the program that remains to be executed also starts with the operator $limit$ as long as the program inside the scope of this operator is not completed.

To provide a better comparison of these recently introduced operators, let us discuss three different programs: δ_1 ; δ_2 and programs with heuristics $local(\delta_1)$; δ_2 and $limit(\delta_1)$; δ_2. We need several new definitions before we can state formally the relation between policies computed using the last two programs.

Recall that according to Definition 5.2.2, a branch B of a policy π is a sequence of actions and test expressions. If an action A in this sequence is stochastic, then it is followed by a sequence of sensing actions defined in *senseEffect*(A), followed by a test expression corresponding to one of nature's choices. In the following definition, we assume that we are given a basic action theory and optimization theory $\mathcal{D}$ that provide axiomatization of an MDP; truth of formulas is understood as entailment from the conjunction of these two theories.

Definition 6.2.1: Let S_1 be a ground reachable situation, B be a branch of a policy π that has a leaf different from *Stop* and $\mathcal{D}$ be a background axiomatization. Then, S_2, a *situation resulting from execution of B starting from S_1* is defined inductively as follows.

1. If B is a deterministic agent action A and $Poss(A, S_1)$ is true, then $S_2 = do(A, S_1)$. If B is *Nil*, then $S_2 = S_1$

2. Let B be a branch $(A\ ;\ B')$, where A is a deterministic agent action and B' is a branch. If $Poss(A, S_1)$ is true and S_2 is a situation resulting from execution of B' starting from $do(A, S_1)$, then S_2 is a situation resulting from execution of B starting from S_1.

3. Let B be a branch $(\alpha\,;\ \textit{senseEffect}(\alpha)\ ;\ \phi?;\ B')$, where B' is a branch, α is a stochastic agent action, *senseEffect*(α) is a sequence of primitive sensing actions $A_1; \ldots; A_l$, ϕ is a test expression that occurs in *senseCond*(N, ϕ) such that N is one of choices available for nature when the agent executes α. If $\phi(do([A_1; \ldots; A_l], do(N, S_1)))$ is true, and S_2 is a situation resulting from execution of B' starting from $do([A_1; \ldots; A_l], do(N, S_1)))$, then S_2 is a situation resulting from execution of B starting from S_1.

4. We say that S_2 is a *situation resulting from execution of a policy π starting in S_1*, if there is a branch of π such that S_2 results from execution of this branch starting from S_1.

We define also when a policy π_2 is a sub-policy of a policy π. Informally speaking, policies are conditional Golog programs of a certain form and one policy is a sub-policy of another, larger policy, if it is a sub-program of this larger program.

Definition 6.2.2: A policy π_2 is a sub-policy of π, if any branch of π_2 is a sub-sequence of a corresponding branch in π. In this case, we say also that π *is an expansion of* π_2.

Note that π may have branches that do not intersect any of the branches in π_2, but if they do intersect, then a branch from π is longer than a corresponding branch from π_2.

The relation between the programs $local(\delta_1)\ ;\ \delta_2$ and $limit(\delta_1)\ ;\ \delta_2$ can be characterized by the following simple observation about policies computed using these two constructs as defined in (6.2) and (6.3), respectively.

Theorem 6.2.3: *Let S be a ground reachable situation, π_1 be a policy computed from δ_1 in S by* IncrBestDo$(\delta_1; Nil, S, h, \gamma, \pi_1, u_1, pr_1)$*, π be a policy that* IncrBestDo$(\pi_1; \delta_2, S, h, \delta_{rem}, \pi, u, pr)$ *computes from $(\pi_1; \delta_2)$ also in S, S_1 be any situation resulting from execution of π_1 starting from S, as defined in (6.2.1), and π_2 be a policy computed by* IncrBestDo$(\delta_2; Nil, S_1, h, \gamma, \pi_1, u_1, pr_1)$ *from δ_2 given the situation S_1. Then, π_2 is a sub-policy of π.*

The proof follows immediately from the above definitions and from an observation that π_1 is a deterministic (conditional) Golog program and for this reason it is a sub-policy of π expanded by policies that will be computed from δ_2 in every situation that results from execution of π_1 starting in S. Because S_1 is just one of the situations that can result from execution of π_1, the policy π_2 is just one of those policies that expand π_1 to π.

Because S_1 is one of (potentially many) situations that can result from on-line execution of the policy π_1, and π_2, according to the definition (6.3) of the $limit$ operator is computed only once in S_1 when execution of π_1 is completed, this theorem indicates that finding out the policy π from $local(\delta_1); \delta_2$ by the off-line interpreter $IncrBestDo$ may require a significant amount of potentially futile computations.

This theorem has an important consequence that we can gain considerable computational savings in cases when π_1 has many leaves and δ_2 is a large non-deterministic program. Indeed, instead of computing off-line all possible continuations from π_1 to π in every situation that may result from execution of π_1 in S, we can simply execute on-line actions dictated by π_1, and once we get to the only situation S_1 that will be the result of this on-line execution, we can compute off-line the policy constrained by δ_2 only once. All other (potential) situations S' that might be the result of executing π_1 in S are no longer relevant and it does not make sense to spend time on computing policies from them. Note also that this successive on-line execution of policies recomputed from $limit(p_i)\,;\,\delta_2$ requires a sequence of computations that takes time depending only on the program δ_1. As an example, assume that a given program δ_1 has occurrences of m nondeterministic choice operators and assume that a policy π_1 computed from δ_1 in S includes n different stochastic actions with 2 possible outcomes (and has no occurrences of deterministic actions). Then, there are 2^n different situations that can be reached by executing the policy π_1 from S_0. An off-line interpreter given $local(\delta_1); \delta_2$ must consider all these 2^n situations when it computes off-line a policy π from $(\pi_1; \delta_2)$. Even in a simple case, when the given program δ_2 mentions only deterministic agent actions, but has k nondeterministic choice operators, then the off-line interpreter has to compare, in total, $2^m \cdot 2^n + 2^n \cdot 2^k$ different branches of the decision tree to determine an optimal policy π. This provides significant computational savings in comparison to $2^m \cdot 2^n \cdot 2^k$ branches that would be compared if the interpreter were given a program $(\delta_1; \delta_2)$ without any bounds on search. However, if given a program δ_1, one computes off-line and executes on-line repeatedly a policy determined by δ_1 and its sub-programs[6], and after completing δ_1, one computes an optimal policy from δ_2, then only $2^{m+1} \cdot (2^n - 1) + 2^k$ comparisons will be required. This demonstrates significant computational savings provided by combination of off-line search with on-line execution in comparison to pure off-line search.

To provide a simple illustration of relative differences between $\delta_1\,;\,\delta_2$ vs. $local(\delta_1)\,;\,\delta_2$ we continue with a delivery example.[7] In particular, we continue with the example considered in Section 5.6, where we assume that there are several people in an office environment, there is mail addressed to each of them and we use the program

while $(\exists person.\neg attempted(person) \wedge \exists n$ *mailPresent(person,n)*$)$
    $local($ $\pi(p:\ people)$
        $(\neg attempted(p) \wedge \exists n$ *mailPresent(p,n)*$)?\ ;\ deliverTo(p)\ )$
    $)$
endWhile

Because this program implements a greedy-like algorithm that chooses a person without looking ahead what deliveries can be made on subsequent iterations, this program yields a policy that has an accumulated expected reward less than an optimal policy in associated MDP. However, in cases where it is important to make decisions quickly, the loss in reward can be

[6]Note that these consecutive computations require in total less than $2^m \cdot \sum_{i=0}^{n-1} 2^{n-i} = 2^{m+1} \cdot (2^n - 1)$ comparisons.

[7]It is more convenient to use programs with *local* than programs with *limit* for comparison with programs without these operators, because former can be used for purely off-line computations and as we just observed the difference between *local* and *limit* becomes important only in the context of on-line executions.

compensated by the decrease in the computation time provided by the *local* operator.

6.3 An on-line interpreter and its implementation

Given the definitions of *IncrBestDo* mentioned in the previous section, we can consider now the on-line interpretation coupled with execution of Golog programs. The definitions translate directly into Prolog clauses. In this section we propose one possible architecture for an on-line interpreter, mention briefly possible variations, but postpone discussion of alternatives to Section 6.5. The following on-line interpreter calls the *incrBestDo(E,S,ER,H,Pol1,U1,Prob1)* interpreter to compute off-line an optimal policy from the given program expression E, gets the first action of the optimal policy, commits to it, executes it in the physical world, and then repeats with the rest ER of the program expression. The following is such an interpreter implemented in Prolog:[8]

```
online(E,S,H,Pol,U) :-   incrBestDo(E,S,H,ER,Pol1,U1,Prob1),
     ( final(ER,S,H,Pol1,U1), Pol=Pol1, U=U1 ;
       reward(V,S),
       ( ER \== nil, Pol1 = (A : Rest) ;  ER == nil, Pol1 = A ),
       (
         (agentAction(A), deterministic(A,S),
            doReally(A),      /* execute A in reality */
            senseExo(do(A,S),Sg),  decrement(H , Hor),
            !,    /* commit to the result */
            online(ER,Sg,Hor,PolRem,URem),
            Pol=(A : PolRem), U $= V + URem
          ) ;
         (
          (senseAction(A),
             doReally(A),     /* do sensing */
             senseExo(do(A,S),Sg),  decrement(H , Hor),
             !,    /* commit to results of sensing */
             online(ER,Sg,Hor,PolRem,URem),
             Pol=(A : PolRem), U $= V + URem
           ) ;
          (agentAction(A), nondetActions(A,S,NatOutcomes),
```

[8]We switch to Prolog because the interpreter uses cuts '!' symbols that are required to prevent backtracking.

```
            doReally(A),     /* execute A in reality */
            senseOutcome(A,S,NatOutcomes,SEff),
                             /* SEff results from real sensing */
            senseExo(SEff,Sg),  decrement(H , Hor),
            !,    /* commit to the result */
            online(ER,Sg,Hor,PolRem,URem),
            Pol=(A : PolRem), U $= V + URem
          )
        )
      )
    ).

% final(Program,Situation,Horizon,Pol,Val) is true iff Program
% cannot be continued because Horizon is zero or if the computed policy
% Pol cannot be executed. Returns value Val.

final(E,S,H,Pol,V) :- H==0, Pol=nil, reward(V,S), VL $<= V, rmax(VL).
final(nil,S,H,Pol,V) :- Pol=nil, reward(V,S), VL $<= V, rmax(VL).
final(stop : E,H,S,Pol,V) :- Pol=stop, reward(V,S), VL $<= V, rmax(VL).
final(E,S,H,Pol,U) :- (Pol == stop ; Pol == nil),
            reward(V,S), VL $<= U, rmax(VL).
```

A complete Prolog code of the implementation is attached in Appendix D.1.[9] Note that in cases when there are no exogenous actions and when there is no need to learn a better probabilistic model of transitions between states, this implementation can be improved by caching the computed policy and reusing it.

The on-line interpreter uses the Prolog cut (!) to prevent backtracking to the predicate $doReally$: we need this because once actions have been actually performed in the physical world, the robot cannot return to the situation that existed before actions were executed.

The on-line interpreter uses the predicate `senseOutcome`$(A, S1, List, S2)$, and the two auxiliary predicates `diffSequence`(A, Seq) and `diagnose`$(S1, S2, List, N)$. We describe below their meaning and show their implementation in Prolog.

The predicates `senseOutcome`$(A, S1, List, S2)$ and `diagnose`$(S1, S2, L, N)$ play the role of combination of *senseEffect*(A) with conditionals that we discussed in the previous chap-

[9]The predicate `rmax(V)` is a built-in predicate in Eclipse Prolog 3.5.2 that maximizes the value of V with respect to the set of linear temporal constraints; operators $<= define some of these linear constraints.

ter. We need them here because a proposed on-line interpreter never gets to execute *senseEffect* from policies computed by the off-line interpreter. This happens simply because the on-line interpreter always takes the very first action from the most recently computed policy and executes it on-line, on the next iteration, it takes again the first action from a new policy, etc.[10] Our on-line interpreter cannot execute the *senseEffect* procedures because on each iteration it computes a new policy from the remaining Golog program that may have no occurrences of *senseEffect* procedures. (Recall that these procedures are inserted only in policies which are a very special case of a general Golog program.) Given the stochastic action A, the list of available outcomes $List$ and the current situation $S1$, the predicate `senseOutcome`$(A, S1, List, S2)$ holds if $S2$ is the situation that results from doing a number of sensing actions necessary to differentiate between all those actions in $List$ that nature can choose to execute. The predicate `diffSequence`(A, Seq) holds if Seq is the sequence of sensing actions $(a_1; a_2; \ldots; a_n)$ specified by the programmer in the domain problem representation. This sequence is differentiating if after doing all actions in the sequence, the action chosen by 'nature' as the outcome of stochastic action A can be uniquely identified. Because each sensing action has as an argument a variable bound to a value provided by a particular sensor, a differentiating sequence of sensing actions results in the situation that can be used by `diagnose`.

```
senseOutcome(A,S,NatureActions,SE) :-  differentiatingSeq(A,Seq),
         getSensorInput(do(X,S),Seq, SE),
      /* find X: an unknown outcome of the stochastic action A */
         diagnose(S,SE,NatureActions,X).

getSensorInput(S,A, do(A,S)) :-  senseAction(A),
    doReally(A).  /* connect to sensors and get data for  */
                  /* a free variable V in A=sense(f,V,Time)*/

getSensorInput(S,(A : Tail),SE) :-  senseAction(A),
       doReally(A),   /* connect to sensors and get data */
       getSensorInput(do(A,S),Tail, SE).
```

The predicate `diagnose`$(S1, S2, L, N)$ takes as its first argument the situation before stochastic action was executed, as its second argument the situation resulting from getting a

[10]Note that we could discard *senseEffect* constructs when we defined the $IncrBestDo$ interpreter, but we decided to keep them to conform with the definition of policy and because alternative architectures of on-line interpreters might wish to use them.

sensory input: it contains 'enough' information to disambiguate between different possible outcomes of the last stochastic action A. The third argument is the list L of all outcomes that nature may choose if the agent executes the stochastic action A in $S1$ and the last argument is the action that nature actually performed. This is the action that we would like to determine: once we know this nature's action, we know the situation that resulted when the agent performed actually the stochastic action A. We can identify which action nature has chosen using the set of mutually exclusive test conditions *senseCond*(n_i, ϕ_i), where ϕ_i is a term representing a situation calculus formula: if ϕ_i holds in the current situation, then we know that nature has chosen the action n_i (n_i belongs to the list L).

```
diagnose(S,SEff,[N],X) :- senseCond(N,C), X=N, holds(C,SEff).

diagnose(S,SEff,[N, NN | Outcomes],X) :-
    senseCondition(N,C), /* assume that the outcome N happened */
    ( X=N, holds(C,SEff) /* verified that nature did action N */ ;
      diagnose(S,SEff,[NN | Outcomes],X) /* if not, consider other outcomes */
     ).
```

Our on-line interpreter includes several auxiliary predicates like `senseExo, decrement`, but names of these predicates are self-explanatory and additional details of their implementations can be found in Appendix D.1. In the following section we consider an example that illustrates how the new construct $limit$ works in the context of the on-line interpreter.

6.4 An Example

In this section we continue to discuss delivery domains where natural constraints on robot behavior can be conveniently expressed in a Golog program (see examples in Section 5.4, Section 5.3 and Example 3.2.1). When the robot is in the main office, it can sense the battery voltage by doing the action $sense(Battery, v, t)$ and, if necessary, connect to the charger. In domains of this type, we can use the on-line interpreter to find and execute incrementally a quasi-optimal policy. In addition to the predicates, function symbols and constants mentioned in the examples above, we use the relational fluent $status(p, st, s)$: a person p can be in or out of his/her office and the robot can determine the current status st when it tries to give an item to the person. The successor state axiom for this fluent:

$$
\begin{aligned}
& status(person, st, do(a, s)) \equiv \\
& \quad (\exists t) a = giveS(item, person, t) \wedge st = In \vee \\
& \quad (\exists t) a = giveF(item, person, t) \wedge st = Out \vee \\
& \qquad status(person, st, s) \wedge \neg(\exists i, p, t) a = giveS(i, p, t) \wedge \neg(\exists i, p, t) a = giveF(i, p, t).
\end{aligned}
$$

In the initial situation, the robot is located in the main office and there are several requests for coffee; the status of all people can be unknown. The MDP problem is to find an optimal policy that corresponds to delivering coffee in the optimal way.

The following compact Golog procedures express natural constraints of the robot's behavior (with abbreviations MO (main office), C (coffee)). The argument r of the procedure *deliver*(r) denotes the finite range of offices and each office is the finite range of people who occupy it; for example, the argument of this procedure can be the finite set of offices:

$$\{LP271, LP276, LP290b, LP269\},$$

where the office $LP271$ is the finite set of names $\{Steve, Maurice, Yves\}$.

proc $deliver(r)$
  while $(\exists x)\, in(x, r) \wedge (\exists person)\, in(person, x) \wedge$
      $(\exists t_1, t_2) wantsCoffee(person, t_1, t_2) \wedge \neg status(person, Out)$ **do**
  $\pi(office : r)$
    $(\ limit(\ \pi(person : office)$
        $(\exists t_1, t_2\ wantsCoffee(person, t_1, t_2) \wedge \neg status(person, Out))?$;
          $\pi t.\ [deliverCoffee(person, t)]$;
            $\pi t.\ [(now(t))?;\ goto(MO, t)]\ ;\ leaveItems\ ));$ /*end of "limit"*/
    $limit(\ \pi(v, t).\ [(now(t))?; sense(Battery, v, t);$
        if $v < 24$ **then** $chargeBattery(t)$ **else** $Nil]$) /*end of "limit"*/
endWhile
endProc

According with the semantics of $\pi(x : \tau)p$ given in Section 6.2, the first occurrence of this construct in the procedure $deliver(r)$ represents the nondeterministic choice between offices given as the argument to the procedure, the second occurrence of this construct represents the nondeterministic choice between people occupying each office. We use the guard $(\exists t_1, t_2)\, wantsCoffee(person, t_1, t_2) \wedge \neg status(person, Out))?$ to consider only those people who want coffee and who are not out of their offices (we assume that initially all people are in their offices, but people may leave and come back at any time, and we do not model this behavior). Note that we use the operator $limit$ to limit the scope of how far the incremental interpreter has to search to find the first person p to serve. Given the program $deliver(r)$, the interpreter considers all offices in r as alternative choices, makes a choice, chooses nondeterministically a person in that office and if the choice satisfies the guard, computes the optimal policy of delivering coffee to a chosen person. Because this last computation occurs inside the scope of the operator $limit$, the interpreter does not look beyond. According to the given reward function and to the given probabilities that people are in their offices, the interpreter will decide who is the optimal person to serve first (it is the person with an early unfulfilled request and who is in her office with a high enough probability). Once this decision has been made, the robot can start executing the procedure $deliverCoffee(p, t)$ at some moment of time t (this moment of time will be determined by product of computing the reward for giving coffee to p). The procedures $deliverCoffee(p, t)$ and $goto(loc, t)$ have been considered previously in Section 5.4. Then, the procedure $deliver(r)$ specifies to sense the voltage of the robot's battery: if the voltage is low, then the battery needs to be charged; otherwise, the computation continues until all requests are served. Finally, the procedure $leaveItems$ is designed to return undelivered items to the main office. If a person was not in his/her office at the time when the robot arrived, then upon the return to the main office the robot carries some items and this procedure commands to put them back:

proc $leaveItems$
  while $(\exists item)\ carrying(item)$ **do**
    $\pi item.\, [(carrying(item))?\ ;$
      $\pi t_N(\ (now(t_N))?\ ;\ putBack(item, t_N)\ )\,].$
endProc

To compare the computation time, we ran both the procedure $deliver(r)$ and its modification that does not include the construct $limit$ (all runs were done on a computer with two Pentium II 300 Mz processors and 128Mb of RAM). More search is required if $deliver$ does not include the $limit$ construct: it takes the incremental interpreter about 1 sec to compute the optimal policy for 2 people, about 12 sec to compute the optimal policy for 3 people and about 2 min to compute the optimal policy for 4 people (note that the decision tree grows exponentially with the number of people). But when we run the procedure $deliver$ with the $limit$ construct, then the computation time remains less than 3 sec even if the argument of $deliver$ is a list of 4 or 5 offices with 7 or 8 people in them in total.

Notice that the program restricts the policy that the robot can implement, leaving only one choice (the choice of person to whom to deliver an item) open to the robot and the rest of the robot's behavior is fixed by the program. However, structuring a program this way may, in general, preclude optimal behavior. For instance, the outer most *limit* construct restricts the search to find 'the best' person to serve at the current time without looking ahead what consequences this choice will have in the future. In circumstances when the robot lingered this may lead the program to skip a next request in favor of a later request. But if the program inside the scope of the outer most *limit* construct would consider all sequences of the next two requests (or the next three requests), then the choice of 'the best' person to serve now would take into account also a next one (or next two, respectively) request(s) to be served in the future and still the computation of the optimal policy would require an acceptable time. In addition, the program can be improved by allowing the robot to carry more than one item at a time: this may lead to better behavior (for instance, in cases when two people from the same office requested different items).

Successful tests of the implementation described here were conducted on the mobile robot B21 in a real office environment. They demonstrated that using the expressive set of Golog operators it is straightforward to encode domain knowledge as constraints on the given large MDP problem. The operator $limit(p)$ proved to be useful in providing heuristics which allowed to compute sub-policies in real time.

6.5 Discussion

An incremental Golog interpreter based on the single-step $Trans$-semantics is introduced in [De Giacomo and Levesque, 1999b]. The Golog programs considered there may include binary sensing actions. The interpreter considered in this chapter is motivated by similar intuitions, but it is based on a different decision-theoretic semantics and employs more expressive representation of sensing.

An approach to integrating planning and execution in stochastic domains [Dearden and Boutilier, 1994] is an interesting alternative to the approach proposed here. The authors suggest to compute a policy close to optimal by searching to a fixed depth of a decision tree and using a heuristic function to estimate the value of states; once a certain probabilistic action is deemed best, it is executed and its actual outcome is observed, then the algorithm repeats.

The online interpreter developed in this chapter allows to consider Golog programs with sensing actions. However, it is based on the assumptions that an MDP describing the planning agent in an environment is fully observable and that probabilities of transitions between states are stationary and do not change with time. If the agent participates in a dynamic game like soccer where individual players do not have a complete probabilistic information about each state of the game, the players may need to constantly monitor their environments by performing frequent sensing actions to check whether previously computed policies should be executed or recomputed. [Ferrein *et al.*, submitted] develops of an alternative on-line DTGolog interpreter suitable for highly dynamic and fast paced environments and reports about successful testing of this interpreter on mobile robots playing soccer. A detailed discussion of this work is available in [Fritz, 2003].

DYNA architectures combine reinforcement learning and execution time planning into an integral computational process that operates alternately on the world and on a learned model of the world [Sutton, 1990, Sutton and Barto, 1998b]. Informally, DYNA architectures learn a world model online (e.g., they can learn transition probabilities) while using approximations to dynamic programming to plan optimal behavior. Our online interpreter can be developed into a DYNA-style architecture by supplementing the incremental interpreter that is responsible for planning with an additional component that gradually builds a more accurate model from interaction with the real world.

The idea of integration of adaptation, responsible for learning a better model of the environment, with control, responsible for computing the best action given the current estimate of the model, is one of the corner stones in the control theory and there are many monographs on adaptive control that explore this idea, e.g. see [Fel'dbaum, 1965, Tsypkin, 1971].

Our on-line interpreter is somewhat related to the idea of *model predictive control* (MPC) or *receding horizon control* (RHC) that received significant attention in the engineering literature. As characterized in the review [Mayne *et al.*, 2000], MPC is "a form of control in which the current control action is obtained by solving on-line, at each sampling instant, a finite horizon open-loop optimal control problem, using the current state of the plant as the initial state; the optimization yields an optimal control sequence and the first control in this sequence is applied to the plant." The authors of this review note that MPC is not a new method of control design, but it is different from other controllers (both for non-linear and linear dynamical systems with constraints) because "it solves the optimal control problem on-line for the current state of the plant, rather than determining off-line a feedback policy (that provides the optimal control for all states)". The main difference between our proposal and the extensive MPC literature is that we are interested here in fully observable MDPs with (potentially very large) finite state and action spaces, but in the majority of engineering systems state and action spaces are subsets of $\mathcal{R}^n$.

In Section 2.4.2, we extensively discussed heuristic search methods (RTA^* and $LRTA^*$) and real-time dynamic programming methods (an envelope-based approach of [Dean *et al.*, 1995] and *RTDP*). Our on-line interpreter is motivated by a similar intention of using heuristic search techniques. More specifically, our on-line DTGolog interpreter is motivated by the idea that planning must be interleaved with execution [Barto *et al.*, 1995, Dearden and Boutilier, 1997]. However, there are significant differences between our approach and the previous ones. The main algorithmic difference is that the previous approaches incrementally improve an estimate of a value function for visited states starting from an heuristic value function because they rely on an asynchronous value iteration algorithm. In [Dearden and Boutilier, 1997], an heuristic function is computed by solving an abstracted MDP with a considerably smaller state space. In addition, off-line decision-tree search is facilitated by several pruning strategies in decision tree construction (e.g., utility pruning and expectation pruning) that provide significant computational savings. In the on-line interpreter proposed in this chapter, values of the current state and of the successor states that can be reached from the current state are discarded once they are computed using the *IncrBestDo* interpreter and these values are not reused in subsequent computations. One of the possible directions for future work is to cache both the policies and values of visited states computed by the *IncrBestDo* interpreter with an eye towards reusing them later. Because our representation of an MDP relies on the situation calculus (i.e., we use a predicate logic based representation in contrast to a state based representation considered in [Barto *et al.*, 1995, Dearden and Boutilier, 1997]), caching may require representing value functions by logical rules. The existing work on asynchronous policy iteration [Sutton, 1990, Williams and Baird, 1993a, Bertsekas and Tsitsiklis, 1996] can also be useful for designing a more elaborate version of an on-line DTGolog interpreter. These further developments will be particularly useful in the adaptive case, when an agent must learn a model of an environment simultaneously with acting in this environment.

There is in depth investigation of on-line algorithms in [Borodin and El-Yaniv, 1998]. The question of which optimization problems can be optimally or approximately solved by "greedy-like" algorithms is studied in [Borodin *et al.*, 2002] where the authors proposed a framework for proving lower bounds on the approximability of a problem by "greedy-like" algorithms.

There is a considerable literature on *macro-actions* and *options*: [Parr and Russell, 1998, Hauskrecht *et al.*, 1998, Parr, 1998b, Parr, 1998a, Sutton *et al.*, 1999, Precup, 2000]; we discussed these papers in the previous chapter. Our approach is different because we consider a high level programming language and domain specification approach: they together provide a convenient tool for computing approximately optimal policies in large-scale MDPs that arise naturally in robotics context. In [Andre and Russell, 2002] and [Pfeffer, 2001], high-level programming languages *ALisp* and *IBAL*, respectively, are proposed for similar problems, but they are not based on a principled framework for reasoning about actions with a solution to the frame problem. For this reason, it can be problematic to integrate them into a larger system that requires logical reasoning about actions.

6.6 Future work

Several important issues are not covered in this chapter. One of them is integration of the on-line interpreter proposed in this chapter with a learning algorithm that gradually improves the model. Another issue is monitoring and rescheduling of policies.

We included in this chapter a discussion of the computational savings that can be achieved by using the constructs $local$ and $limit$ in Golog programs. Our robotics implementation also demonstrates that the $limit$ construct is useful in providing heuristics which allowed us to compute sub-policies in real time (without this construct search would be computationally infeasible in the office delivery application). This preliminary evidence indicates that it would be interesting in the future to do more experimental work on the evaluation of the online interpreter and the specific constructs $limit$ and $local$. This work can be conducted on MDP problems of a moderate size where an optimal policy can be computed by standard dynamic programming algorithms and, as a consequence, both values of optimal policies (for selected states) and computation time (required to compute an optimal policy) can be recorded for comparison. In addition, this work should be done on benchmark domains where an MDP can be decomposed into several independent sub-problems such that a Golog programmer can take advantage of this additional structure by using constructs $limit$ or $local$ to indicate which problems can be solved independently and how their solutions can be combined. Furthermore, the on-line interpreter can be used to compute optimal policies in the set of policies constrained by a Golog program that uses either $local$ or $limit$ operator. Again, values of computed polices for an initial state and computation time can be recorded for comparison. We hope that in some realistic domains there will be natural constraints on an MDP such that even if there will be a moderate loss in the expected values of policies computed using our on-line interpreter, this loss will be compensated by significant savings in the time required to compute policies with respect to the time required to compute them using exact algorithms. Because in the case of $limit$, off-line computations must be interleaved with on-line executions, it will be necessary to factor out the time required to execute actions (in reality or on simulator) to account for computation time only (this is especially important if rewards depend on the time when the last action is executed).

A very interesting direction for future work is designing a more sophisticated on-line DT-Golog interpreter that employs sampling, learning and pruning [Kearns *et al.*, 1999, Ballard, 1983, Sutton, 1990, Dearden and Boutilier, 1997]. Another direction is caching expected values and policies computed by the interpreter with intention to reuse them in subsequent computations.

The diagnostic task that the current version of the on-line interpreter solves is admittedly oversimplified. We expect that additional research on integrating the on-line incremental interpreter with the diagnosis approach proposed in [McIlraith, 1998] will allow us to formulate a more comprehensive version.

Chapter 7

Conclusion

In this chapter we provide a brief summary of the predicate logic-based frameworks developed in this thesis in the context of research conducted in Cognitive Robotics and mention the main contributions of our thesis.

7.1 Summary

In the last 25 years, many researchers working in AI have raised serious concerns about feasibility of using logic-based approaches for the purposes of designing controllers for robots. Existing approaches were criticized for a lack of computational efficiency, a lack of real-time performance in realistic domains and for not taking seriously the complexity of the real world environments, including exogenous actions and events, and uncertainty in sensor information and in robot's actions. However, with the advent of the Cognitive Robotics approach some of these concerns have been successfully addressed. In particular, the researchers working at the University of Toronto, York University (Toronto, Canada), the University of Rome ”La Sapienza” the University of Aachen (Germany) and other universities have developed several versions of the predicate logic-based high level programming language Golog. These developments are based on encoding a large amount of declarative information about the world in terms of Reiter's basic action theories and on providing the robot (or a software agent) with procedural knowledge in terms of a high level program written in a Golog family programming language. There are also a number of alternative approaches that have been developed using different logical frameworks, e.g., see [Sandewall, 1995, Sandewall, 1998, Shanahan, 1997, Shanahan, 1998, Poole, 1998, Doherty, 1999, Baral and Gelfond, 2000, Baral and Son, 2001, Thielscher, 2000b, Thielscher, 2000a, Amir, 2001, Grosskreutz, 2002].

The work reported in this thesis builds on the conceptual framework developed by the Cognitive Robotics group in the University of Toronto. Our thesis demonstrates that reliable and flexible controllers for mobile robots can be successfully and conveniently designed using situation calculus-based logical theories augmented with procedural knowledge represented in a Golog program. In particular, our work addresses issues related to taking into account uncertainty and incomplete information about an environment. We proposed and explored two different perspectives to dealing with uncertain and complex environments. First, we developed a logical framework that can (under certain conditions) provide reliable functioning of a robotics system without using probabilistic information about the environment. Second, we developed a situation calculus based approach to designing reliable and conveniently programmable controllers in the case when there is probabilistic information about the environment and functioning of the robot can be characterized as a fully observable MDP. Because in both cases the robot must get a feedback from the environment, we proposed and explored a computationally efficient approach to reasoning about sensory information.

7.2 Contributions

The main contributions of this thesis are the following.

- We developed logical representations and frameworks that allow the robot to perform reasoning fast in realistic scenarios. This is achieved by approximating a general logic reasoning task by a more specialized reasoning task. More specifically, in reasoning about sensing, we proposed an approach that is sound with respect to a more general K-based one, but our approach sacrifices completeness to gain a computational efficiency. In the context of the decision-theoretic planning, we also proposed and explored an approach that sacrifices optimality of a policy to gain a better computational performance.

- We proposed a general Golog-based framework for execution monitoring, and explored two different realizations of this framework that guarantee a successful recovery of Golog programs from unexpected failures. In the case when the high level control module gets only temporal information as the sole sensory input from the external world, we developed a particular execution monitor that was successfully tested using the mobile robot B21 on the realistic scenarios. Our logical framework turned out to be useful for rescheduling and modification of temporal plans in the presence of unpredictable disturbances in the environment. From the robotics point of view, we developed a general architecture suitable for monitoring execution of restartable situation-calculus based programs written in Golog.

- We have provided a general first-order language for specifying MDPs and imposing constraints on the space of allowable policies by writing a program. In this way we have provided a natural framework for combining decision-theoretic planning and agent programming with an intuitive semantics. We have found this framework to be very flexible as a robot programming tool, integrating programming and planning seamlessly, and permitting the developer to choose the point on this spectrum best-suited to the task at hand.

- We proposed and explored an on-line decision-theoretic interpreter that combines off-line computation of policies optimal with respect to a current model of the world with online execution of initial actions from these policies. We argued that the proposed interpreter can lead to significant computational savings in comparison to a purely off-line decision-theoretic interpreter.

Our thesis is mainly concerned with developing efficient controllers for mobile robots, but we conjecture that potential areas of applicability are much larger.

7.3 Future work

In this section we would like to summarize future work from previous chapters.

The work on execution monitoring considered in Chapter 4, can be continued in several possible directions. First, it would be interesting to overcome some limiting assumptions like the assumption that all exogenous actions are malicious and design a recovery procedure that can take advantage of serendipitous events. Second, in cases when the goal of a program can be analysed into more fine-grained sub-goals, it would be interesting to see how this analysis can contribute to managing runtime violations of requirements. Finally, it is important to take into account temporal constraints on computations performed by the recovery mechanism on-line and do these computations in any-time mode (the more time is available, the better recovery program can be computed).

The work on DTGolog (Chapter 5 and Chapter 6) can also be extended in many possible directions. We are planning to integrate efficient algorithms and other techniques for solving MDPs into this framework (dynamic programming, pruning, sampling, etc.) Other possible future avenues include: incorporating realistic models of partial observability (a key to ensuring wider applicability of the model); extending the expressive power of the language to include concurrency and other advanced features; incorporating declaratively-specified heuristic and search control information. It would be interesting to do more extensive experimental analysis of the DTGolog approach using different realistic domains both in the context of robotics and other applications. It would be very interesting to integrate the on-line interpreter with a learning component to extend this approach to cases when a model of the environment must be learned concurrently with acting in the environment.

Appendix A

An Execution Monitor with A Planner As A Recovery Mechanism

A.1 An Interpreter

An implementation of $DoEM$ considered below is simpler than the relation *DoEM*(δ, exo, s, h, s_f) specified in Section 4.2 because this implementation does not uses traces.

```
/*************************************************************************/
/*        Golog Interpreter With A Simple Execution Monitor.           */
/*        This version does not keep trace of program states.          */
/*     Based on Trans-interpreter by DeGiacomo, Lesprance, Levesque.   */
/*************************************************************************/
:- set_flag(print_depth,100).
:- pragma(debug).
:- set_flag(all_dynamic, on).

:- 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.*/

/* trans(Prog,Sit,Prog_r,Sit_r) */

trans(A,S,nil,do(A,S)) :- primitive_action(A), poss(A,S).

trans(?(C),S,nil,S) :- holds(C,S).

trans(P1 : P2,S,P2r,Sr) :- final(P1,S),trans(P2,S,P2r,Sr).
trans(P1 : P2,S, P1r : P2,Sr) :- trans(P1,S,P1r,Sr).

trans(P1 # P2,S,Pr,Sr) :- trans(P1,S,Pr,Sr) ; trans(P2,S,Pr,Sr).
```

```
trans(pi(V,P),S,Pr,Sr) :- sub(V,_,P,PP), trans(PP,S,Pr,Sr).

trans(star(P),S,PP : star(P),Sr) :- trans(P,S,PP,Sr).

trans(if(C,P1,P2),S,Pr,Sr) :-
     trans(((?(C) : P1) # (?(-C) : P2)),S,Pr,Sr).

trans(while(C,P),S,Pr,Sr) :-
     trans(star(?(C) : P) : ?(-C), S,Pr,Sr).

trans(P,S,Pr,Sr) :- proc(P,Pbody), trans(Pbody,S,Pr,Sr).


/* final(Prog,Sit) */

final(nil,S).

final(P1 : P2,S) :- final(P1,S),final(P2,S).

final(P1 # P2,S) :- final(P1,S) ; final(P2,S).

final(pi(V,P),S) :- final(P,S).

final(star(P),S).

final(if(C,P1,P2),S) :-
      final(( (?(C) : P1) # (?(-C) : P2) ),  S).

final(while(C,P),S) :-
      final( star(?(C) : P) : ?(-C) ,S).

final(pcall(P_Args),S) :- proc(P_Args,P),final(P,S).

/* transCL(Prog,S,Prog_r,S_r) is the transitive closure
                                of trans(Prog,S,Prog_r,S_r) */
transCL(P,S,P,S).
transCL(P,S,Pr,Sr) :- trans(P,S,PP,SS), transCL(PP,SS,Pr,Sr).

/* offline(Prog,S,Sr): Sr is the situation resulting after doing Prog */
offline(Prog,S0,Sf) :- final(Prog,S0), S0=Sf;
               trans(Prog,S0,Prog1,S1),
               offline(Prog1,S1,Sf).

doEM(Prog,S0,Sf) :-  final(Prog,S0), S0=Sf;
                    trans(Prog,S0,Prog1,S1),
                   offline(Prog1,S1,Sg), /*  Prog1 leads to the goal */
                     !,
```

```
                   monitor(Prog1,S1,Prog2,S2), !,
                   doEM(Prog2,S2,Sf).

/* 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 non fluents, including
   Prolog system predicates. For this to work properly, the GOLOG programmer
   must provide, for all fluents, a clause giving the result of restoring
   situation arguments to situation-suppressed terms, for example:
         restoreSitArg(ontable(X),S,ontable(X,S)).             */

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)).

restoreSitArg(poss(A),S,poss(A,S)).
```

A.2 An Execution Monitor

An implementation of $Monitor$, and implementations of $Relevant$ and $Recover$ considered below have been simplified with respect to the formal specifications in Section 4.3 because this implementation does not use traces and exogenous actions are typed in by an user.

```
/*********************************************************************/
/*          The execution monitor calls a recovering procedure        */
/*            to counter-balance external disturbances.               */
/*          The recovery procedure uses a simple planner.             */
/*********************************************************************/
/* :- nodbgcomp.   */
:- pragma(debug).

/* We allow EXOs be arbitrary Golog programs; hence, if Exo happens
   in S, then S1 is the situation resulting after termination of Exo */

monitor(DeltaOld,S,DeltaNew,S1) :- printMessages(DeltaOld,S),
       sense(Exo,S),
        ( Exo=noOp -> (S1=S, DeltaNew=DeltaOld); /*No exogenous disturbances */
             /* Otherwise, simulate execution of exogenous */
          offline(Exo,S,S1),
             /* Exogenous program cannot be redone */
          !,
          ( relevant(DeltaOld,S,S1) ->
                            ( write('Start recovering...'), nl,
                              recover(1,4,DeltaOld,S1,DeltaNew)
                             );
             /* otherwise, continue */
             nl, write(' No recovery is necessary. '),
             write('Proceeding with the next step of program. '), nl,
             DeltaNew=DeltaOld
            )
         ).


/* Determine whether the remainder of the program will
    terminate in the goal situation  */

relevant(DeltaOld,S,S1) :-
            not offline(DeltaOld,S1,Sg).


printMessages(Delta,S) :- nl, write('  Program state = '), write(Delta), nl,
     write('  Current situation: '), write(S), nl.

sense(E,S):-
nl, write('>Enter: an exogenous program or noOp if none occurs.'),
```

```
           nl, read(E1),

 /* IF an exogenous is  noOp,  or a terminating (hence, legal) Golog program
   THEN  this simulator will consider it as just happened.  */

( (E1 = noOp; offline(E1,S,Sres)) -> E=E1 ;

 /* ELSE  print error message, and try again.  */

    write(">> Your program is not possible. Try again."), nl, sense(E,S)).


recover(M,N,PrOld,S,PrNew) :- M =< N, straightLineProgram(Plan,S,M),
offline(Plan : PrOld, S, Sg),   /* Can we recover using Plan? */
PrNew=(Plan : PrOld),      /* Yes: prefix old program by Plan */
nl,
write('  New program = '), write(PrNew),
nl;
/* Otherwise, try to find a longer sequence of actions */
M < N,
/* Complexity bound is not exceeded yet  */
M_inc is M+1,
recover(M_inc,N,PrOld,S,PrNew).

recover(M,N,PrOld,S,PrNew) :- M > N, nl,
write('-------------------------'),
write('|  Recovery procedure FAILED  |'),
write('-------------------------'),nl,
PrNew=nil.

straightLineProgram(Plan,S,K) :- K >= 1,
 primitive_action(A), poss(A,S),
goodAction(Plan,A),
( K =:= 1  -> Plan=A ;
  K_dec is K -1,
  straightLineProgram(TailPlan,do(A,S),K_dec),
    Plan=(A : TailPlan)
).

/* Elaborate in the future:
A is a good action to perform after Plan if A does not undo
the results of the last action in Plan. Otherwise, A is futile.
*/
goodAction(Plan,A).

/*--------------------- An execution example ----------------------
```

```
Example below demonstrates that exos can be counterbalanced up to
the moment when  ?(goal) was not yet performed.

>Enter: an exogenous program or noOp if none occurs.
move(s7,r1).
Start recovering...

  New program = moveToTable(s7) : nil : ?(goal)

  Program state = nil : nil : ?(goal)
  Current situation: do(moveToTable(s7), do(move(s7, r1), do(move(r1, o1),
do(moveToTable(r1), do(move(r1, o1), do(move(o1, m1), do(move(m1, e1),
do(move(f, r2), do(move(r2, n), do(moveToTable(n), do(moveToTable(s7),
do(moveToTable(r2), do(move(r2, s7), do(move(s7, n), do(move(n, e1),
s0)))))))))))))))

>Enter: an exogenous program or noOp if none occurs.
noOp.

  Program state = nil
  Current situation: do(moveToTable(s7), do(move(s7, r1), do(move(r1, o1),
do(moveToTable(r1), do(move(r1, o1), do(move(o1, m1), do(move(m1, e1),
do(move(f, r2), do(move(r2, n), do(moveToTable(n), do(moveToTable(s7),
do(moveToTable(r2), do(move(r2, s7), do(move(s7, n), do(move(n, e1),
s0)))))))))))))))

>Enter: an exogenous program or noOp if none occurs.
moveToTable(r1).

 No recovery is necessary. Proceeding with the next step of program.

yes.
*/
```

A.3 A Blocks World Example

```
:- set_flag(print_depth,100).
:- pragma(debug).
/* :- nodbgcomp. */
:- set_flag(all_dynamic, on).
:- dynamic(p/1).

/* This program has to build (non-deterministically) one of the two towers
                r               p
                o               a
                m      or       r
                e               i
                                s
given several blocks with letters  r, o, m, e, p, a, r, i.
There is no block with ``p''.
Final positions of other blocks can be arbitrary.   */

/*  Primitive Action Declarations  */

primitive_action(moveToTable(X)).
primitive_action(move(X,Y)).

/*  Action Precondition Axioms  */

poss(move(X,Y),S) :- clear(X,S), clear(Y,S), not X = Y.
poss(moveToTable(X),S) :- clear(X,S), not ontable(X,S).


/*  Successor State Axioms  */

on(X,Y,do(A,S)) :- A = move(X,Y) ;
                   on(X,Y,S), A \= moveToTable(X), A \= move(X,Z).

ontable(X,do(A,S)) :- A = moveToTable(X) ;
                      ontable(X,S), A \= move(X,Y).

clear(X,do(A,S)) :- (A = move(Y,Z) ; A = moveToTable(Y)), on(Y,X,S) ;
                    clear(X,S), A \= move(Y,X).

/* Initial Situation: everything is on table and clear.    */
r(r1).   r(r2).             o(o1).  o(o2).  o(o3).
m(m1). m(m2).               e(e1).  e(e2).

/* There is no block with the letter "p"  */
a(a1).  i(i1).  i(i2).   s(s7).

ontable(r1,s0).  ontable(r2,s0).
```

```
ontable(o1,s0).  ontable(o2,s0).  ontable(o3,s0).
ontable(m1,s0).  ontable(m2,s0).
ontable(e1,s0).  ontable(e2,s0).
ontable(s7,s0).
ontable(a1,s0).
ontable(i1,s0).  ontable(i2,s0).
ontable(n,s0).  ontable(f,s0).

clear(r1,s0).  clear(r2,s0).
clear(o1,s0).  clear(o2,s0).  clear(o3,s0).
clear(m1,s0).  clear(m2,s0).
clear(e1,s0).  clear(e2,s0).
clear(s7,s0).
clear(a1,s0).
clear(i1,s0).  clear(i2,s0).
clear(n,s0).
clear(f,s0).

build :- doEM((tower : ?(goal)), s0, S).

proc(tower,
             makeParis # makeRome).

proc(makeParis,
    pi(b0, ?(s(b0) & ontable(b0) & clear(b0)) :
        pi(b1, ?(i(b1)) : move(b1,b0) :
            pi(b2, ?(r(b2)) : move(b2,b1) :
                pi(b3, ?(a(b3)) : move(b3,b2) :
                    pi(b4, ?(p(b4)) : move(b4,b3))))))).

proc(makeRome,
    pi(b0, ?(e(b0) & ontable(b0) & clear(b0)) :
        pi(b1, ?(m(b1)) : move(b1,b0) :
            pi(b2, ?(o(b2)) : move(b2,b1) :
                pi(b3, ?(r(b3)) : move(b3,b2)))))).


goal(S) :-  p(Y1), a(Y2), r(Y3), i(Y4), s(Y5), ontable(Y5,S),
            on(Y4,Y5,S), on(Y3,Y4,S), on(Y2,Y3,S), on(Y1,Y2,S), clear(Y1,S);
            r(X1), o(X2), m(X3), e(X4),
            ontable(X4,S), on(X3,X4,S), on(X2,X3,S), on(X1,X2,S), clear(X1,S).


  restoreSitArg(ontable(X),S,ontable(X,S)).
  restoreSitArg(on(X,Y),S,on(X,Y,S)).
  restoreSitArg(clear(X),S,clear(X,S)).
  restoreSitArg(goal,S,goal(S)).
```

Appendix B

An Execution Monitor with Backtracking As A Recovery Mechanism

B.1 An Interpreter for Temporal Golog Programs

```
/*********************************************************************
  A Golog interpreter with preferences, rewards and utilities based on
  Trans-interpreter by DeGiacomo, Lesprance, Levesque, IJCAI-1997
  and the idea of sequential, temporal Golog by Reiter, KR-1998
*********************************************************************/
:- lib(r).
:- lib(fromonto).
:- set_flag(print_depth,300).
:- pragma(debug).
/* :- nodbgcomp.     */
:- set_flag(all_dynamic, on).
:- dynamic(proc/2).
:- set_flag(all_dynamic, on).

:- 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.*/

                    /* GOLOG  clauses. */

/* trans(Prog,Sit,Prog_r,Sit_r) */

trans(P1 : P2,S,P2r,Sr) :- final(P1,S),trans(P2,S,P2r,Sr).
trans(P1 : P2,S, P1r : P2,Sr) :- trans(P1,S,P1r,Sr).
trans(?(C),S,nil,S) :- holds(C,S).
```

```
trans(P1 # P2,S,Pr,Sr) :- trans(P1,S,Pr,Sr) ; trans(P2,S,Pr,Sr).

trans(pi(V,P),S,Pr,Sr) :- sub(V,_,P,PP), trans(PP,S,Pr,Sr).

trans(star(P),S,PP : star(P),Sr) :- trans(P,S,PP,Sr).

trans(if(C,P1,P2),S, Pr,Sr) :-
            holds(C,S),  trans(P1,S,Pr,Sr) ;
            holds(-C,S), trans(P2,S,Pr,Sr ).

trans(while(C,P),S,Pr,Sr) :- holds(-C,S), Pr=nil, Sr=S;
            holds(C,S),
            trans(P : while(C,P),S, Pr,Sr).

trans(P,S,Pr,Sr) :- proc(P,Pbody), trans(Pbody,S,Pr,Sr).

trans(A,S,nil,do(A,S)) :- primitive_action(A), deterministic(A), poss(A,S),
                          start(S,T1), time(A,T2), T1 $<= T2.

/* NEW GOLOG CONSTRUCT: pick the best value for a variable.
  Beware that be(V,E) fails if there is no execution of E1
   (E with an instantiated V) leading to a situation S1 such
  that pref(S1,S,S) (i.e., S1 better than the current situation S).
  But if an execution of be(V,E) terminates it results in a
  better situation than the current situation S.             */

trans(be(V,E),S,Pr,Sr) :- sub(V,_,E,E1), trans(E1,S, Pr,Sr),
                 bestSit(Pr,Sr, Sbest),
                 not dominated(be(V,E),S,E1,Sbest).

bestSit(E,S,Best) :-  offline(E,S,Best),
                 not ( offline(E,S,Sf), pref(Sf,Best,S) ),
                 pref(Best,S,S).

dominated(be(V,E),S,E1,S1) :- sub(V,_,E,E2),
                 bestSit(E2,S,S2),
                 E2 \= E1,
                 pref(S2,S1,S).

/* NEW GOLOG CONSTRUCT: pick a good value for a variable.
  Beware that an execution of good(V,E) may result in a
  situation S1 such that pref(S1,S,S) is false (i.e., S1
  is not better than the current situation S). But if
  there is a terminating execution of E1 (E with an
  instantiated V), then good(V,E) terminates too.          */

trans(good(V,E),S, Pr,Sr) :- sub(V,_,E,E1),
```

```
                 trans(E1,S, Pr,Sr),
                 offline(Pr,Sr, Sgood ),
                 not ( offline(E1,S,S12), pref(S12,Sgood,S) ),
                 not better(good(V,E),S,E1,Sgood).

better(good(V,E),S,E1,S1) :- sub(V,_,E,E2),
                 offline(E2,S,S2), E2 \= E1,
                 not ( offline(E2,S,S21), pref(S21,S2,S) ),
                 pref(S2,S1,S).


/* final(Prog,Sit) */

final(nil,S).
final(fail,S).
final(P1 : P2,S) :- final(P1,S),final(P2,S).
final(P1 # P2,S) :- final(P1,S) ; final(P2,S).
final(pi(V,P),S) :- final(P,S).
final(star(P),S).
final(P_Args,S) :- proc(P_Args,Pbody), final(Pbody,S).


/* transCL(Pr,S,Pr_r,S_r) is the transitive closure of trans(Pr,S,Pr_r,S_r) */

transCL(P,S,P,S).
transCL(P,S,Pr,Sr) :- trans(P,S,PP,SS), transCL(PP,SS,Pr,Sr).

/* offline(Pr,S,Sr): Sr is the situation resulting after doing Pr in S */

offline(Prog,S0,Sf) :- final(Prog,S0), Sf=S0;
               trans(Prog,S0,Prog1,S1),
               offline(Prog1,S1,Sf).

bestTrans(Delta,S,Delta2,S2) :-
                 trans(Delta,S,Delta2,S2),
                 offline(Delta2,S2,Best),
                 not ( offline(Delta,S,Sf), pref(Sf,Best,S) ),
                 %       pref(Best,S,S),
                 util(Ut, Best), U $=< Ut, rmax(U),
                 current_stream(reports, X, Stream),
                 printf(Stream, "\n
          >> Current program = %w\n
          >> Next Program = %w\n
          >> BestPlan = %w \n
          >> Utility = %w \n", [Delta,Delta2,Best,U]).
```

```
     /* Preferences and rewards */

/* pref(S1,S2,S): the situation S1 is better than S2 wrt S.
  On the situation tree, S is a predecessor of both S1 and S2 */

/* A simple implementation of preferance relation by means of utilities */

pref(S1, S2, S) :-  util(Val1,S1), V1 $=< Val1,
              util(Val2,S2), V2 $=< Val2,
              rmax(V1), rmax(V2),
              !,   /* no backtracking */
              V1 > V2.

%   util(V,S) is a fluent: V is a total utility in situation S.
%   See an example of utility function in the file  "coffee"


                    /* Time specific clauses.  */

start(do(A,S),T) :- time(A,T).
/*  start( s0, 0 ).  Say this in the file coffee  */

/* "now" is a synonym for "start". */
now(S,T) :- start(S,T).

/*  schedMin(S1, Sg)  determines a schedule such that
    times of occurrences are minimized. */

schedMin(S1,S1).
schedMin(S1,do(A,S2)) :-
                 S1==S2,
                 time(A,T), (nonground(T) -> rmin(T); true).
schedMin(S1,do(A,S2)) :-
                S1\==S2, schedMin(S1,S2),
                time(A,T), (nonground(T) -> rmin(T); true).


/* 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 non fluents, including
   Prolog system predicates. For this to work properly, the GOLOG programmer
   must provide, for all fluents, a clause giving the result of restoring
   situation arguments to situation-suppressed terms, for example:
         restoreSitArg(ontable(X),S,ontable(X,S)).            */

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)).

restoreSitArg(poss(A),S,poss(A,S)).
restoreSitArg(start(T),S,start(S,T)).
restoreSitArg(now(T),S,now(S,T)).
restoreSitArg(currentSit(Sc), S, currentSit(Sc,S)).
restoreSitArg(util(V), S, util(V,S)).
restoreSitArg(pref(S1,S2), S, pref(S1,S2,S)).

currentSit(S1,S2) :- S1=S2.
```

B.2 An Execution Monitor

```
/*******************************************************************
   Monitor of Temporal Sequential Golog Programs with Rewards.
  This requires a Golog interpreter with preferences and rewards.
 ******************************************************************/

/* online(Prog,Trace,S0,Sf):  Sf is the final situation after doing Prog startin
  from S0 where Trace is the sequence of states the program already passed by
  (e.g., when it did non-deterministic branching).   */

online(Delta,H,S,Sf) :-
        final(Delta,S), Sf=S;
        bestTrans(Delta,S,Delta1,S1),
             %  trans(Delta,S,Delta1,S1), offline(Delta1,S1,Sgoal),
        H1 = trace(Delta,H),
   %     (S1\==S ->  offline(Delta1,S1,Sgoal);  S1==S),
   %     nl, write(">> Delta1= "), write(Delta1), nl,
   %         write(">> BestPlan= "), write(Sgoal), nl,
   %         write(">> Utility= "), util(V,Sgoal), write(V), nl,
         schedMin(S,S1),     /* Schedule the next action */
           !,              /* do not allow backtracking */
        monitor(S,Delta1,S1,H1,Delta2,S2,H2),
            !,                     /* do not allow backtracking */
        online(Delta2,H2,S2,Sf).


/*  So far,  no recovery if initial situation is bad */

goalAchievable(Program, S1, Sg) :-
     offline(Program, S1, Sg),   /* Program leads to a goal situation.*/
     schedMin(S1, Sg).           /* and there is a schedule */

/*
schedMin(S1,S1).
schedMin(S1,do(A,S2)) :-
                 S1==S2,
                 time(A,T), (nonground(T) -> rmin(T); true).
schedMin(S1,do(A,S2)) :-
                S1\==S2, schedMin(S1,S2),
                time(A,T), (nonground(T) -> rmin(T); true).
*/

monitor(S,Delta1,S1,Trace1,Delta2,S2,Trace2) :-
       S == S1,          /* skip tests: they do not have temporal arg*/
       Delta2 = Delta1,
       S2 = S1,
       Trace2 = Trace1.
```

```
monitor(S,Delta1,S1,Trace1,Delta2,S2,Trace2) :-
       S1=do(A, S),      /* find what action is selected for execution */
       nl,
       write(">> action "), write(A), write(" is planned for execution"),
       nl,
       sense(RealTime, S),    /* watch the current time */
       write(">> The current time is "), write(RealTime),
       Se=do(watch_time(RealTime),S),
       time(A, ScheduledTime),
       replaceTime(ScheduledTime, RealTime, Delta1, NewDelta1),
       ( relevant(NewDelta1,S1,Se) ->
             recover(S1,NewDelta1,Trace1,Se,Delta2,Trace2,S2);
             runAction(A,Se,S2), Delta2=NewDelta1, Trace2 = Trace1
        ).


/* If action is belated, it should be rescheduled according to a time clock
  because actions cannot be executed in the past. Note that sensed time
  may be relevant wrt execution of Delta1 _only if_  action is belated.
  Otherwise, it is sufficient to wait until the time when the action
  was scheduled originally and then do the action in reality.   */

relevant(NewDelta,S1,Se) :-
         S1= do(Action, S),      /* what action has to be done? */
         start(Se, RealTime),
         time(Action, ScheduledTime),
         ScheduledTime < RealTime,
         /* Yes, the action is belated */
         nl, write(">> RELEVANT is working ..."), nl,
         replaceTime(ScheduledTime, RealTime, Action, A),
         !,
         not offline(A : NewDelta, Se, Sg).


replaceTime(OldTime, NewTime, OldPr, NewPr) :-
          sub(OldTime, NewTime, OldPr, Prog) -> NewPr= Prog;
          NewPr= OldPr.    /* If OldPr does not mention OldTime */

recover(S1,Delta1,History1,Se1,Delta2,History2,S2) :-
       sense(RealTime, Se1),    /* watch the current time */
       write(">> The current time is "), write(RealTime),
       Se2=do(watch_time(RealTime),Se1),
   recover1(S1,Delta1,History1,Se2,Delta2,History2,S2) ;
       sense(RealTime, Se1),    /* watch the current time */
       write(">> The current time is "), write(RealTime),
       Se2=do(watch_time(RealTime),Se1),
```

```
   recover2(S1,Delta1,History1,Se2,Delta2,History2,S2).


recover1(S1,Delta1,History1,Se1,Delta2,History2,S2) :-
        nl, write(">> RECOVER1 is working..."), nl,
          sense(RealTime, Se1),    /* watch the current time */
          write(">> The current time is "), write(RealTime),
          Se2=do(watch_time(RealTime),Se1),
        History1 \== empty,
        History1 = trace(Delta, OldHistory),
        ( S1= do(Action, S),
          time(Action, ScheduledT),
          replaceTime(ScheduledT, RealTime, Delta, NewDelta),
    /* Can we skip Action and the subsequent actions
       and start anew ? */
          offline(NewDelta, Se2, Sg), nl,
          printf(">> Skip %w and subsequent actions\n", [Action]),
          Delta2 = NewDelta, History2= OldHistory, S2=Se2,
          current_stream(reports, X, Stream),
          printf(Stream, ">> Skip %w and subsequent actions\n", [Action]),
          printf(Stream, "\n >> Recovered program = %w\n
                            >> New Situation = %w \n", [Delta2,S2]) ;
          recover1(S1,Delta1,OldHistory,Se2,Delta2,H2,S2) ).


recover2(S1,Delta1,History1,Se1,Delta2,History2,S2) :-
        nl, write(">> RECOVER2 is working..."), nl,
          sense(RealTime, Se1),    /* watch the current time */
          write(">> The current time is "), write(RealTime),
          Se2=do(watch_time(RealTime),Se1),
        History1 \== empty,
        History1 = trace(Delta, OldHistory),
        ( S1= do(Action, S),
          time(Action, ScheduledT),
          replaceTime(ScheduledT, RealTime, Delta, NewDelta),
          replaceTime(ScheduledT, RealTime, Action, NewAction),
  /* Can we do NewAction, skip the subsequent actions and start anew ? */
          offline(NewAction : NewDelta, Se2, Sg), nl,
   printf(">> Do %w , but skip subsequent actions\n", [NewAction]),
          runAction(NewAction, Se2, S2),
          Delta2 = NewDelta, History2= OldHistory,
          current_stream(reports, X, Stream),
          printf(Stream, ">> Skip %w and subsequent actions\n", [Action]),
          printf(Stream, "\n >> Recovered program = %w\n
                            >> New Situation = %w \n", [Delta2,S2]) ;
          recover2(S1,Delta1,OldHistory,Se2,Delta2,H2,S2) ).
```

```
recover(S1,Delta1,History1,Se,Delta2,History2,S2) :-
Delta2=fail, History2 = empty, S2=Se,
nl,
write(" Program cannot be recovered. "),
write(" Monitor terminates its execution."), nl.


/* The following clause of "runAction" respects the schedule selected by
  the program:  ScheduledTime is the earliest time the action  A
  can start in the current situation Se under the condition that
  the remaining program Delta1 will lead to a successful termination.
  If the current time Time is less than ScheduledTime, wait until
  ScheduledTime, and then start executing.                     */

runAction(A,Se,S2) :-
        time(A, ScheduledTime),  /* the time to start A */
        start(Se, Time),
             Time =< ScheduledTime,
        nl, write(" runAction is working"), nl,
        PassTime is ScheduledTime - Time,
        ( PassTime > 0 -> myWait(PassTime,Se); true),
        execute(A, ScheduledTime),    /* do action now in reality ! */
        S2 = do(A, Se),
        current_stream(reports, X, Stream),
        printf(Stream, "\n>> New situation = %w\n", [S2]),
        printf("\n >> New Situation = %w\n", [S2]).

/*
        util(V,S2), rmax(V),
        printf("\n>> Situation = %w\n>> Utility = %w\n", [S2,V]),
        current_stream(reports, X, Stream),
        printf(Stream, "\n>> Situation = %w\n>> Utility = %w\n", [S2,V]).
*/

/* The following clause of "runAction" reschedules the action A
  if it is belated, but the delay is NOT relevant with respect
  to the execution of remaining program.
*/

runAction(A,Se,S2) :-
        time(A, ScheduledT),       /* the time to start A */
        start(Se, RealTime),
            RealTime > ScheduledT,
        nl, write(" runAction is working"), nl,
        replaceTime(ScheduledT,RealTime, A, Act),  /* reschedule action */
        execute(Act, RealTime),    /* do A now in reality ! */
```

```
        S2=do(Act, Se),
        current_stream(reports, X, Stream),
        printf(Stream, "\n>> New situation = %w\n", [S2]),
        printf("\n >> New Situation = %w\n", [S2]).

/*
        util(V,S2), rmax(V),
        printf("\n>> Situation = %w\n>> Utility = %w\n", [S2,V]),
        current_stream(reports, X, Stream),
        printf(Stream, "\n>> Situation = %w\n>> Utility = %w\n", [S2,V]).
*/

/* time sensing on robot  */
sense(Time, S):-
            get_flag(unix_time, UnixTime),
            read(UnixStart) from_file scratch,
            Time is UnixTime - UnixStart.


/* time sensing in simulation
sense(Time, S):-
            nl, write('>> Enter: current time.'), nl,
 read(NewTime),
            start(S, OldTime),
            ( OldTime > NewTime -> sense(Time, S);
             Time = NewTime ).
*/

/*-------------------------------------------------------*
*                      Tools                            *
*-------------------------------------------------------*/

setTime(A) :- time(A, Var), (nonground(Var) -> rmin(Var); true).


primitive_action(watch_time(RealTime)).
time(watch_time(RealTime), RealTime).

/*  RUN  STUFF  */

coffee(S) :- online(deliverOneCoffee(3),empty, s0, S).
visit(S) :- online(serve,empty,s0,S).
```

B.3 Motivating Examples

This section includes examples that we use to motivate backtracking as a recovery procedure. We run all examples using the following wrapper file:

```
:- [ 'golog' ].
:- [ 'moni' ].
:- [ 'axioms' ].
%  :- [ 'coffee' ].          % calls  hli_go_path()
%  :- [ '~mes/bee/bin_solaris/hli-eclipse.pl' ].  % interface to BeeSoft
:- [ 'exec' ].

run :-
      open( reports, append, Stream),
      get_flag(unix_time, StartT),    % Unix time (in seconds)
      write(StartT) onto_file scratch, % store start time
      printf(Stream, "\n\n   This report is started at time %w\n", [StartT]),
/* IF */   ( online( serveNew(20) : ?(goal), empty, s0, LastSit ) ->
/* THEN */     ( nl, write(">> The last situation is "),
                 write(LastSit) ), nl ;
/* ELSE */   printf( "Sorry, something is wrong...\n", [] ),
            flush( output )
           ),
          get_flag(unix_time, EndT),
          Elapsed is EndT - StartT,
          printf(" Time elapsed is %w seconds\n", [Elapsed]),
          printf(Stream, "Time elapsed is %w seconds\n", [Elapsed]),
          flush( output ),
          write(LastSit) onto_file history,
          printf(Stream, "The last situation is \n %w \n", [LastSit]),
          close(Stream),
          wait( 10 ),
          hli_go( 160, 2530).

wait( Time ) :-
        nl, printf( "*-> wait for %w seconds\n", [ Time ] ), nl,
        flush(output),
        doWait( Time ).

vis1 :-  online( visit1, empty, s0, Sfin).
vis2 :-  online( visit2, empty, s0, Sfin).
```

The file “axioms” includes all precondition and successor state axioms. The file “exec” includes auxiliary (execution related) predicates. In addition, running examples requires Golog code of one of the examples together with the desription of the initial situation. The next two files include Golog code of examples: the file “visit1” includes the Golog code of the first example, and the file “visit2” includes the Golog code of the second example.

B.3.1 Axioms for examples that illustrate backtracking.

```
:- set_flag(all_dynamic, on).
:- dynamic(proc/2).

/* GOLOG Procedures */

proc(goto(L,T),
  pi(rloc,?(robotLocation(rloc)) : pi(deltat,?(travelTime(rloc,L,deltat)) :
     goBetween(rloc,L,deltat,T)))).

proc(goBetween(Loc1,Loc2,Delta,T),
   ?(Loc1=Loc2 & Delta=0) #
         ?(-(Loc1=Loc2) & Delta > 0) : startGo(Loc1,Loc2,T) :
   pi(t, ?(t $= T + Delta) : endGo(Loc1,Loc2,t)) ).

/* Preconditions for Primitive Actions */

poss(pickupCoffee(T),S) :- not holdingCoffee(S), robotLocation(cm,S),
                   start(S,TS),  TS $<= T.

poss(giveCoffee(Person,T),S) :- holdingCoffee(S),
                                robotLocation(office(Person),S),
                                start(S,TS), TS $<= T.

poss(startGo(Loc1,Loc2,T),S) :- not going(L,LL,S),
                                robotLocation(Loc1,S),
                                start(S,TS), TS $<= T.

poss(endGo(Loc1,Loc2,T),S) :- going(Loc1,Loc2,S),
                                start(S,TS), TS $<= T.


/* Successor State Axioms */

hasCoffee(Person,do(A,S)) :- A = giveCoffee(Person,T) ;
                             hasCoffee(Person,S).

robotLocation(Loc,do(A,S)) :- A = endGo(Loc1,Loc,T) ;
                      ( robotLocation(Loc,S),
                       not A = endGo(Loc2,Loc3,T) ).

going(Loc1,Loc2,do(A,S)) :- A = startGo(Loc1,Loc2,T) ;
                                  (going(Loc1,Loc2,S),
                                   not A = endGo(Loc1,Loc2,T)).

holdingCoffee(do(A,S)) :- A = pickupCoffee(T) ;
                                (holdingCoffee(S),
```

```
                                 not A = giveCoffee(Person,T)).

util(0, s0).
util(V2, do(giveCoffee(Person,T),S)) :- util(V, S),
             wantsCoffee(Person,T1,T2), not hasCoffee(Person,S),
               V1 $<= (T2 - T)/2,
              V1 $<= T - (3*T1 - T2)/2 ,
              V2 $= 700 - T + V + V1*11/10.

util(V, do(A,S)) :- A \=giveCoffee(P,T), util(V, S).

/* The time of an action occurrence is its last argument. */

time(pickupCoffee(T),T).
time(giveCoffee(Person,T),T).
time(startGo(Loc1,Loc2,T),T).
time(endGo(Loc1,Loc2,T),T).

/* Restore situation arguments to fluents. */

restoreSitArg(robotLocation(Rloc),S,robotLocation(Rloc,S)).
restoreSitArg(hasCoffee(Person),S,hasCoffee(Person,S)).
restoreSitArg(going(Loc1,Loc2),S,going(Loc1,Loc2,S)).
restoreSitArg(holdingCoffee,S,holdingCoffee(S)).

/* Primitive Action Declarations */

primitive_action(pickupCoffee(T)).
primitive_action(giveCoffee(Person,T)).
primitive_action(startGo(Loc1,Loc2,T)).
primitive_action(endGo(Loc1,Loc2,T)).


travelTime(L,L,0).
travelTime(L1,L2,T) :- travelTime0(L1,L2,T) ;
                       travelTime0(L2,L1,T).

travelTime0(cm,office(sue),15).
travelTime0(cm,office(mary),10).
travelTime0(cm,office(bill),8).
travelTime0(cm,office(joe),10).
travelTime0(office(bill),office(sue),18).
travelTime0(office(bill),office(mary),15).
travelTime0(office(sue),office(mary),5).

drive( StartPos, EndPos ) :- nl, write("drive, drive, drive from "),
                             write(StartPos), write(" to "), write(EndPos), nl.
```

```
/*
The file ``simul'' is the wrapper that calls the golog interpreter,
the monitor of temporal programs, the file ``coffee'' that defines
the application and the file ``exec'' that includes auxilary predicates.

:- [ 'golog' ].
:- [ 'moni' ].
:- [ 'coffee' ].          % calls  hli_go_path()
:- [ 'exec' ].
:- set_flag(recover/7, spy, on).
:- set_flag(recover/7, leash, stop).

run :-     online( deliverCoffee(2), empty, s0, LastSit ),
           schedMin(s0, LastSit),
           nl, write(">> The last situation is "),
                 write(LastSit), !, nl.

run :-     printf( "Sorry, something is wrong...\n", [] ),
            flush( output ).

visit1 :-  online( visit1, empty, s0, Sfin).
visit2 :-  online( visit2, empty, s0, Sfin).
visit3 :-  online( visit3, empty, s0, Sfin).
*/
/*
dvp.cs> eclipse -b simul
structures.pl compiled traceable 3764 bytes in 0.03 seconds
suspend.pl compiled traceable 8012 bytes in 0.13 seconds
r.sd       loaded traceable 61220 bytes in 0.48 seconds
fromonto.pl compiled traceable 3304 bytes in 0.01 seconds
golog      compiled traceable 19856 bytes in 0.62 seconds
moni       compiled traceable 13668 bytes in 0.06 seconds
coffee     compiled traceable 45472 bytes in 0.13 seconds
exec       compiled traceable 5000 bytes in 0.04 seconds
simul      compiled traceable 980 bytes in 0.88 seconds
ECRC Common Logic Programming System [sepia opium megalog parallel]
Version 3.5.2, Copyright ECRC GmbH, Wed Jan  3 12:54 1996
[eclipse 1]: visit2.
lists.pl   compiled traceable 8200 bytes in 0.04 seconds


>> action startGo(office(joe), cm, 91) is planned for execution

>> Enter: current time.
91.
>> The current time is 91
```

```
 runAction is working

>> Im doing the action startGo(office(joe), cm, 91) at time 91
  *-> start going from office(joe) to cm

drive, drive, drive from office(joe) to cm

>> action endGo(office(joe), cm, 101) is planned for execution

>> Enter: current time.
120.
>> The current time is 120
>> RELEVANT is working ...


>> RECOVER1 is working...

>> RECOVER1 is working...

>> RECOVER1 is working...

>> RECOVER1 is working...

>> RECOVER1 is working...

>> RECOVER1 is working...

>> RECOVER1 is working...

>> RECOVER1 is working...

>> RECOVER2 is working...

>> RECOVER2 is working...

>> RECOVER2 is working...

>> RECOVER2 is working...

>> RECOVER2 is working...

>> RECOVER2 is working...

>> RECOVER2 is working...

>> Do endGo(office(joe), cm, 120) , but skip subsequent actions
```

```
 runAction is working

>> Im doing the action endGo(office(joe), cm, 120) at time 120
  *-> end going from office(joe) to cm

>> action pickupCoffee(120) is planned for execution

>> Enter: current time.
121.
>> The current time is 121
>> RELEVANT is working ...

 runAction is working

>> Im doing the action pickupCoffee(121) at time 121
  *-> give some coffee please

>> action startGo(cm, office(mary), 121) is planned for execution

>> Enter: current time.
122.
>> The current time is 122
>> RELEVANT is working ...

 runAction is working

>> Im doing the action startGo(cm, office(mary), 122) at time 122
  *-> start going from cm to office(mary)

drive, drive, drive from cm to office(mary)

>> action endGo(cm, office(mary), 132) is planned for execution

>> Enter: current time.
133.
>> The current time is 133
>> RELEVANT is working ...

 runAction is working

>> Im doing the action endGo(cm, office(mary), 133) at time 133
  *-> end going from cm to office(mary)

yes.
*/
```

B.3.2 Execution related predicates.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  EXECUTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Use this version to run examples and simulation */
/*
execute(A,T) :-
     nl, write('>> Im doing the action '), write(A),
     write(' at time '), write(T), nl,
     doReally(A).

wait( Time ) :-
        nl, printf( "*-> wait for %w seconds\n", [ Time ] ), nl,
        flush(output).
*/

/* Use this version to run the robot in the real corridor */

execute(A,T) :-
     nl, write('>> Im doing the action '), write(A),
     write(' at time '), write(T), nl,
        doReally(A),
        current_stream(reports, X, Stream),
        printf(Stream, "\n >> Im doing the action %w at time %w\n", [A,T]).

myWait( Time, S ) :-
        Time > 10,
        sense(BeginTime, S),    /* watch the time before waiting */
        current_stream(reports, X, Stream),
        printf(Stream, "       *-> Time before waiting = %w \n", [ BeginTime ] )
        nl, printf( "*-> start waiting for %w seconds...\n", [ Time ] ), nl,
        flush(output),
        printf(Stream, "       *-> start waiting for %w seconds... \n", [ Time ]
        doWait( Time ),
        sense(EndTime, S),    /* watch the time after waiting */
        DiffTime is EndTime - BeginTime,
        ( DiffTime < Time -> myWait(DiffTime, S) ; true).

/*
   We do not wait for less than 10 seconds. However, waiting for
  small amounts of time can be included, see the next clause below. */

myWait( Time, S ) :-
        nl,
      printf( "*-> I'm too busy: no time to wait for %w seconds\n", [ Time ] ),
        flush(output),
```

```
        sense(EndTime, S),    /* watch the time after waiting */
        current_stream(reports, X, Stream),
        printf(Stream, "       *-> Time after waiting = %w \n", [ EndTime ] ).


myWait( Time, S ) :-  Time < 0,
            /* remove  Time < 0,  to allow waiting for small amount of time */
        sense(BeginTime, S),    /* watch the time before waiting */
        nl, printf( "*-> ok, wait for %w seconds ?\n", [ Time ] ), nl,
        flush(output),
        doWait( Time ),
        sense(EndTime, S),    /* watch the time after waiting */
        DiffTime is EndTime - BeginTime,
        nl, printf( "*-> I waited for %w seconds, indeed\n", [ DiffTime ] ), nl,
        flush(output).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

doReally( pickupCoffee( T ) ) :-
printf( "  *-> give some coffee please at time %w\n", [T] ),
        flush(output).

doReally( giveCoffee( Person, T ) ) :-
printf( "  *-> %w, take your coffee please at time %w\n", [Person,T] ),
        flush(output),
takeYourCoffee.

doReally( startGo( From, To, T ) ) :-
printf( "  *-> start going from %w to %w at time %w\n", [From, To, T] ),
        flush(output),
        driveReal( From, To ).

doReally( endGo( From, To, T) ) :-
printf( "  *-> end going from %w to %w at time %w\n", [From, To, T] ),
        flush(output).

doReally( Action ) :-
printf( "  *-> error of action %w\n", [ Action ] ),
        flush(output).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*  askForCoffee :-
hli_sound_stop,
%     hli_show_picture("give"),
hli_sound("some_coffee"),
```

```
hli_buttons( 3, 3, 3, 3, 20, Answer ),
( Answer=0 ->
    askForCoffee
;
    hli_show_picture("thanks"),
    hli_sound("thank_you")
).
*/
askForCoffee.

/*  takeYourCoffee :-
hli_sound_stop,
%     hli_show_picture("take"),
hli_sound("take_coffee"),
hli_buttons( 3, 3, 3, 3, 20, Answer ),
( Answer=0 ->
    takeYourCoffee
;
    hli_show_picture("thanks"),
    hli_sound("thank_you")
).
*/
takeYourCoffee.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TOOLS: hacks to fight with bugs in Eclipse Prolog.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

doWait( Time ) :-
        Wait is Time*2 +1,
        doWait2( Wait ).

doWait2( Time ) :-
        Time =< 0.
doWait2( Time ) :-
        Time2 is Time-1,
        sleep( 0.5 ),
        doWait2( Time2 ).
```

```
proc(visit1,
     goto(office(mary),1) :
       pi(t1, ?(now(t1)) :
             (  (goto(cm,t1) : pi(t2, ?(now(t2)) : goto(office(sue),t2)))
               #
                (goto(cm,t1) : pi(t2, ?(now(t2)) : goto(office(bill),t2)))
             )
          ) :
         pi(t, ?(now(t)) : ?(t $< 40)) ).

/* Initial Situation (visit1) */
robotLocation(cm,s0).
start(s0, 0).

wantsCoffee(sue,140,160).
wantsCoffee(joe,90,100).
wantsCoffee(bill,100,110).
wantsCoffee(mary,130,170).

travelTime0(cm,office(sue),15).
travelTime0(cm,office(mary),10).
travelTime0(cm,office(bill),8).
travelTime0(cm,office(joe),10).
travelTime0(office(bill),office(sue),18).
travelTime0(office(bill),office(mary),15).
travelTime0(office(sue),office(mary),5).

drive( StartPos, EndPos ) :- nl, write("drive, drive, drive from "),
                      write(StartPos), write(" to "), write(EndPos), nl.
```

```
proc(visit2,
     pi(p, pi(t1, pi(t2, ?(-hasCoffee(p) & wantsCoffee(p,t1,t2)) :
        goto(cm,91) :  pi(t, ?(now(t)) : pickupCoffee(t)) :
                pi(t, ?(now(t)) : goto(office(p),t)) :
                pi(t, ?(now(t)) : ?(t $<= t2) )
       )))
    ).


/* Initial situation (visit2)  */
robotLocation(office(joe),s0).
start(s0, 91).

wantsCoffee(bill,100,110).
wantsCoffee(mary,130,170).
wantsCoffee(sue,140,160).
wantsCoffee(joe,90,100).

travelTime0(cm,office(sue),15).
travelTime0(cm,office(mary),10).
travelTime0(cm,office(bill),8).
travelTime0(cm,office(joe),10).
travelTime0(office(bill),office(sue),18).
travelTime0(office(bill),office(mary),15).
travelTime0(office(sue),office(mary),5).

drive( StartPos, EndPos ) :- nl, write("drive, drive, drive from "),
                    write(StartPos), write(" to "), write(EndPos), nl.
```

B.4 A Coffee Delivery Robot.

```
:- set_flag(all_dynamic, on).
:- dynamic(proc/2).

/*         A Coffee Delivery Robot           */
/* GOLOG Procedures */

/* The coffee serving procedures serve(T) and serveOneCoffee(T)
   (written in regular Golog) below deliver coffee to all people
   who can be served in time.  serve(T) terminates when there
   does NOT exist a person who can be served in his/her
   preferable time. The termination is conditioned only upon
   temporal inequalities. The procedure timeGoal expresses this
   termination condition in regular Golog.                        */

proc(timeGoal,
     pi(t, ?(now(t)) :
       pi(rloc, ?(robotLocation(rloc)) :
         ?(-some(p, some(t1, some(t2,
           some(travTime1, some(travTime2, some(wait,
             wantsCoffee(p,t1,t2) & -hasCoffee(p) &
               wait $>= 0 & travelTime(rloc,cm,travTime1) &
                 travelTime(cm,office(p),travTime2) &
                   t1 $<= t + wait + travTime1 + travTime2 &
                     t + wait + travTime1 + travTime2 $<= t2
      )))))))))
    ).

proc(serve(T),
     timeGoal
   #
     if( holdingCoffee,
          serveOneCoffeeNew(T),
          goto(cm,T) : pi(t, ?(now(t)) : pickupCoffee(t) : serveOneCoffee(t) )
       )
   ).

proc(serveOneCoffee(T),
    pi(p, pi(wait, pi(t1, pi(t2, pi(travTime, pi(rloc,  pi(currTime,
     ?(wantsCoffee(p,t1,t2) & -hasCoffee(p) & (wait $>= 0) & now(currTime) &
         robotLocation(rloc) & travelTime(rloc,office(p),travTime) &
       t1 $<= T + wait + travTime & T + wait + travTime $<= t2) ))))) :
     pi(t, ?(t $= T + wait) : goto(office(p),t)) ) :
     pi(t, ?(now(t)) : giveCoffee(p,t) )
      ) :
     pi(t, ?(now(t)) : serve(t))
    ).
```

```
goal(s0).
goal(do(A,S)) :- A \= giveCoffee(P,T), goal(S).
goal(do(A,S)) :- A = giveCoffee(P,T), pref(do(A,S),S,s0), goal(S).

proc(visit1,
     goto(office(mary),1) :
       pi(t1, ?(now(t1)) :
              (  (goto(cm,t1) : pi(t2, ?(now(t2)) : goto(office(sue),t2)))
                #
                 (goto(cm,t1) : pi(t2, ?(now(t2)) : goto(office(bill),t2)))
              )
          ) :
         pi(t, ?(now(t)) : ?(t $< 40)) ).


proc(visit2,
     pi(p, pi(t1, pi(t2, ?(-hasCoffee(p) & wantsCoffee(p,t1,t2)) :
        goto(cm,91) :  pi(t, ?(now(t)) : pickupCoffee(t)) :
                pi(t, ?(now(t)) : goto(office(p),t)) :
                pi(t, ?(now(t)) : ?(t $<= t2) )
       )))
    ).

proc(goto(L,T),
  pi(rloc,?(robotLocation(rloc)) : pi(deltat,?(travelTime(rloc,L,deltat)) :
     goBetween(rloc,L,deltat,T)))).

proc(goBetween(Loc1,Loc2,Delta,T),
   ?(Loc1=Loc2 & Delta=0) #
          ?(-(Loc1=Loc2) & Delta > 0) : startGo(Loc1,Loc2,T) :
   pi(t, ?(t $= T + Delta) : endGo(Loc1,Loc2,t)) ).

/* Preconditions for Primitive Actions */

poss(pickupCoffee(T),S) :- not holdingCoffee(S), robotLocation(cm,S),
                    start(S,TS),  TS $<= T.

poss(giveCoffee(Person,T),S) :- holdingCoffee(S),
                                robotLocation(office(Person),S),
                                start(S,TS), TS $<= T.

poss(startGo(Loc1,Loc2,T),S) :- not going(L,LL,S),
                                robotLocation(Loc1,S),
                                start(S,TS), TS $<= T.

poss(endGo(Loc1,Loc2,T),S) :- going(Loc1,Loc2,S),
```

```
                                start(S,TS), TS $<= T.

/* Successor State Axioms */

hasCoffee(Person,do(A,S)) :- A = giveCoffee(Person,T) ;
                             hasCoffee(Person,S).

robotLocation(Loc,do(A,S)) :- A = endGo(Loc1,Loc,T) ;
                      ( robotLocation(Loc,S),
                       not A = endGo(Loc2,Loc3,T) ).

going(Loc1,Loc2,do(A,S)) :- A = startGo(Loc1,Loc2,T) ;
                                   (going(Loc1,Loc2,S),
                                    not A = endGo(Loc1,Loc2,T)).

holdingCoffee(do(A,S)) :- A = pickupCoffee(T) ;
                                (holdingCoffee(S),
                                 not A = giveCoffee(Person,T)).

util(0, s0).
util(V2, do(giveCoffee(Person,T),S)) :- util(V, S),
               wantsCoffee(Person,T1,T2), not hasCoffee(Person,S),
               V1 $<= (T2 - T)/2,
               V1 $<= T - (3*T1 - T2)/2,
               V2 $= V + V1.

util(V, do(A,S)) :- A \=giveCoffee(P,T), util(V, S).


/* The time of an action occurrence is its last argument. */
time(pickupCoffee(T),T).
time(giveCoffee(Person,T),T).
time(startGo(Loc1,Loc2,T),T).
time(endGo(Loc1,Loc2,T),T).

/* Restore situation arguments to fluents. */
restoreSitArg(robotLocation(Rloc),S,robotLocation(Rloc,S)).
restoreSitArg(hasCoffee(Person),S,hasCoffee(Person,S)).
restoreSitArg(going(Loc1,Loc2),S,going(Loc1,Loc2,S)).
restoreSitArg(holdingCoffee,S,holdingCoffee(S)).
restoreSitArg(goal,S,goal(S)).

/* Primitive Action Declarations */
primitive_action(pickupCoffee(T)).
primitive_action(giveCoffee(Person,T)).
primitive_action(startGo(Loc1,Loc2,T)).
primitive_action(endGo(Loc1,Loc2,T)).
```

```
/* Initial Situation.  */

robotLocation(park,s0).
start(s0, 0).

wantsCoffee( lounge,    600, 700 ).   % (3*660 - 740)/2 = 620
wantsCoffee( ray,       360, 440 ).  %  (3*460 - 540/2 = 420
wantsCoffee( yves,   160, 240 ).    % (3*220 - 320)/2 = 170

travelTime(L,L,0).
travelTime(L1,L2,T) :- travelTime0(L1,L2,T) ;
                       travelTime0(L2,L1,T).

/*
--- TABLE OF AVERAGE TRAVEL TIME FROM cm ---

        -----------RAY    75  (51, 72 cm->ray; 56, 55 ray->cm)
        |
        +--------------
        |
        |
        +--ITRC            24
    CM--+
        |
        |
        |
        +--VISITOR         40
        |
 LOUNGE-+                 58 (cm->lounge), 63 (lounge->cm)
        |
  ------+--------------------
*/

travelTime0( park,   cm,      100 ).     /* 73, 82 on simulator */
travelTime0( cm,    office(ray),    120 ).   /* 51,72,56,55 on simulator*/
travelTime0( cm,    office(itrc),   30 ).   /* 24 on simulator */
travelTime0( cm,    office(yves),   45 ).   /* 29 on simulator  */
travelTime0( cm,    office(lounge), 75 ).   /* 58, 63 on simulator */

/* Other travel times (measured on simulator):  */

travelTime0( office(ray),    office(itrc),    70 ).
travelTime0( office(ray),    office(yves),  120 ).
travelTime0( office(ray),    office(lounge),  120 ).
travelTime0( office(lounge),  office(itrc),    70 ).
```

```
travelTime0( office(lounge),  office(yves),  60 ).
travelTime0( office(yves),  office(itrc),    50 ).
travelTime0( park,        office(yves),  60 ).

/* Geometric coordinates on the real office map. */
/* office(ray)= (3753.4, 1800)    cm= (2675, 2555)    park= (118, 2487) */

drivePath( cm,    office(lounge),  [ ( 840, 2560 ), ( 770, 2530 ) ] ).
drivePath( cm,    office(yves), [ ( 1620, 2500 ) ] ).
drivePath( cm,    office(ray),
                      [ (3535, 2490 ), (3760, 2480), ( 3770, 1770 ) ] ).

drivePath( office(ray),  cm,
      [ ( 3760, 2420 ), (3560, 2500 ), ( 2820, 2560 ), ( 2690, 2550 ) ] ).
drivePath( park,    cm,      [ ( 2685, 2500 ), ( 2700, 2550 ) ] ).
drivePath( office(yves), cm,
      [ ( 3120, 2450 ), ( 2700, 2550 ) ] ).
drivePath( _, cm,      [ ( 2700, 2550 ) ] ).
drivePath( office(yves), office(ray),
      [ ( 3120, 2450 ), (3535, 2490 ), (3760, 2480), ( 3770, 1770 ) ] ).
drivePath( office(ray),  office(lounge),
      [ ( 3760, 2420 ), (3560, 2480 ), ( 840, 2560 ), ( 770, 2530 ) ] ).

drivePath( X, Z, BigList) :- drivePath( X, Y, List1),
                      drivePath( Y, Z, List2),
                      append(List1, List2, BigList).

driveSim( StartPos, EndPos ) :- nl, write("drive, drive, drive from "),
                       write(StartPos), write(" to "), write(EndPos), nl.

/* Drive in the corridor and turn to the goal location */
driveReal( StartPos, EndPos ) :-
drivePath( StartPos, EndPos, Path ),   % get path
           hli_go_path( Path ),              % drive, drive...
look( EndPos, X, Y ),             % get aim point for turning
hli_turn_to_point( X, Y ).         % and turn the robot

look( office(lounge),  640, 2670 ).
look( office(yves), 1600, 2180 ).
look( cm,      2720, 2720 ).
look( office(itrc),    2872, 2200 ).
look( office(ray),     3728, 1650 ).
```

Appendix C

An Offline Decision-Theoretic Golog

C.1 An Interpreter

```
/*******************************************************************
   An Offline Decision Theoretic Golog Interpreter (non-temporal version)
                        October, 1999.

            Do not distribute without permission.
            Include this notice in any copy made.

Permission to use, copy, and modify this software and its documentation
for non-commercial research purpose is hereby granted without fee,
provided that this permission notice appears in all copies. This
software cannot be used for commercial purposes without written permission.
This software is provided "as is" without express or implied warranty
(including all implied warranties of merchantability and fitness).
No liability is implied for any damages resulting from
or in connection with the use or performance of this software.

The following paper provides the technical background:
@inproceedings{AAAI,
   author = {Boutilier, C. and  Reiter, R. and
             Soutchanski, M. and Thrun, S.},
  title={Decision-Theoretic, High-level Robot Programming in
          the Situation Calculus},
  booktitle = {Proc. of the 17th National Conference on
    Artificial Intelligence (AAAI'00)},
  address={Austin, Texas},
  publisher = {Available at:  http://www.cs.toronto.edu/~cogrobo },
  year= 2000
}

E-mail questions about the interpreter to Mikhail Soutchanski:
```

```
                      mes [at] cs [dot] toronto [dot] edu

NOTICE: this software works on Unix/Linux machines with Eclipse Prolog that is
        available from IC-PARK (Imperial College, London, UK). It is easy to
        modify this interpreter to work with any other version of Prolog.

**********************************************************************/

:- dynamic(proc/2).            /* Compiler directives. Be sure   */
:- set_flag(all_dynamic, on).     /* that you load this file first! */
:- set_flag(print_depth,500).
:- pragma(debug).

:- 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 */

/* The predicate bp() is the top level call. Add an end-of-program
   marker "nil" to the tail of program expression E , then compute
   the best policy that succeeds with probability Prob. The horizon
   H must be a non-negative integer number.
*/


bp(E,H,Pol,Util,Prob,File) :- integer(H), H >= 0,
      cputime(StartT),
      bestDo(E : nil,s0,H,Pol,Val,Prob),
      cputime(EndT), Elapsed is EndT - StartT,
      Util is float(Val),
      open( File, append, Stream),
      date(Date),
      printf(Stream, "\n\n    This report is started at time %w\n", [Date]),
      ( proc(E,Body) ->
        printf(Stream, "The Golog program is\n proc(%w,\n  %w)\n",[E,Body]) ;
        printf(Stream, "The Golog program is\n %w\n",[E])
       ),
      printf("\nThe computation took %w seconds",[Elapsed]),
      printf(Stream, "\nTime elapsed is %w seconds\n", [Elapsed]),
       printf(Stream, "The optimal policy is \n %w \n", [Pol]),
       printf(Stream, "The value of the optimal policy is %w\n",[Util]),
    printf(Stream, "The probability of successful termination is %w\n",[Prob]),
       close(Stream).
```

```
/*  bestDo(E,S,H,Pol,V,Prob)
  Given a Golog program E and situation S find a policy Pol of the highest
  expected utility Val. The optimal policy covers a set of alternative histories
  with the total probability Prob. H is a given finite horizon.
*/

bestDo((E1 : E2) : E,S,H,Pol,V,Prob) :-   H >= 0,
           bestDo(E1 : (E2 : E),S,H,Pol,V,Prob).

bestDo(?(C) : E,S,H,Pol,V,Prob) :-   H >= 0,
          holds(C,S) ->  bestDo(E,S,H,Pol,V,Prob) ;
                ( (Prob is 0.0) , Pol = stop, reward(V,S) ).

bestDo((E1 # E2) : E,S,H,Pol,V,Prob) :-  H >= 0,
     bestDo(E1 : E,S,H,Pol1,V1,Prob1),
     bestDo(E2 : E,S,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).

bestDo(if(C,E1,E2) : E,S,H,Pol,V,Prob) :-   H >= 0,
               holds(C,S) -> bestDo(E1 : E,S,H,Pol,V,Prob) ;
                             bestDo(E2 : E,S,H,Pol,V,Prob).

bestDo(while(C,E1) : E,S,H,Pol,V,Prob) :-   H >= 0,
          holds(-C,S) -> bestDo(E,S,H,Pol,V,Prob) ;
                         bestDo(E1 : while(C,E1) : E,S,H,Pol,V,Prob).

bestDo(ProcName : E,S,H,Pol,V,Prob) :-   H >= 0,
              proc(ProcName,Body),
              bestDo(Body : E,S,H,Pol,V,Prob).

/* Non-decision theoretic version of pi: pick a fresh value of X
   and for this value do the complex action E1 followed by E.
*/
bestDo(pi(X,E1) : E,S,H,Pol,V,Prob) :-   H >= 0,
            sub(X,_,E1,E1_X), bestDo(E1_X : E,S,H,Pol,V,Prob).

/*
  Discrete version of pi. pickBest(x,f,e) means: choose the best value
   of x from the finite non-empty range of values f, and for this x,
   do the complex action expression e.
*/
bestDo(pickBest(X,F,E) : EF,S,H,Pol,V,Prob) :-  H >= 0,
        range(F,R),
        ( R=[D],    sub(X,D,E,E_D),
                     bestDo(E_D : EF,S,H,Pol,V,Prob)          ;
          R=[D1,D2], sub(X,D1,E,E_D1), sub(X,D2,E,E_D2),
```

```
                  bestDo((E_D1 # E_D2) : EF,S,H,Pol,V,Prob)  ;
           R=[D1,D2 | Tail], Tail = [D3 | Rest],
                 sub(X,D1,E,E_D1), sub(X,D2,E,E_D2),
           bestDo((E_D1 # E_D2 # pickBest(X,Tail,E)) : EF,S,H,Pol,V,Prob)
          ).

bestDo(A : E,S,H,Pol,V,Prob) :-   H > 0,
      agentAction(A), deterministic(A,S),
      ( not poss(A,S), Pol = stop, (Prob is 0.0) , reward(V,S) ;
        poss(A,S), Hor is H - 1,
        bestDo(E,do(A,S),Hor,RestPol,VF,Prob),
        reward(R,S),
        V is R + VF,
        ( RestPol = nil,     Pol = A ;
          not RestPol=nil,   Pol = (A : RestPol)
         )
       ).

bestDo(A : E,S,H,Pol,V,Prob) :-   H > 0,
       agentAction(A), nondetActions(A,S,NatOutcomesList),
       Hor is H -1,
       bestDoAux(NatOutcomesList,E,S,Hor,RestPol,VF,Prob),
       reward(R,S),
       V is R + VF,
       Pol=(A : senseEffect(A) : (RestPol)).

bestDoAux([N1],E,S,H,Pol,V,Prob) :-   H >= 0, senseCondition(N1,Phi1),
      ( not poss(N1,S), ( Pol= ?(Phi1) : stop, (Prob is 0.0) , V is 0 ) ;
        poss(N1,S),
        prob(N1,Pr1,S),
        bestDo(E,do(N1,S),H,Pol1,V1,Prob1), !,
        Pol = ( ?(Phi1) : Pol1 ),
        V is Pr1*V1,
        Prob is Pr1*Prob1 ).

bestDoAux([N1 | OtherOutcomes],E,S,H,Pol,V,Prob) :-  H >= 0,
      OtherOutcomes = [Head | Tail],  % there is at least one other outcome
      ( not poss(N1,S) -> bestDoAux(OtherOutcomes,E,S,H,Pol,V,Prob) ;
        poss(N1,S),
        bestDoAux(OtherOutcomes,E,S,H,PolT,VT,ProbT),
        senseCondition(N1,Phi1),
        prob(N1,Pr1,S),
        bestDo(E,do(N1,S),H,Pol1,V1,Prob1), !,
        Pol = if(Phi1,  % then
                             Pol1,  % else
                                       PolT),
        V is VT + Pr1*V1,
```

```
       Prob is ProbT + Pr1*Prob1 ).


bestDo(nil,S,H,Pol,V,Prob) :-
                Pol=nil, reward(V,S), (Prob is 1.0) .

bestDo(nil : E,S,H,Pol,V,Prob) :- H > 0, bestDo(E,S,H,Pol,V,Prob).

bestDo(stop : E,S,H,Pol,V,Prob) :-
                Pol=stop, reward(V,S), (Prob is 0.0) .

bestDo(E,S,H,Pol,V,Prob) :- H =:= 0,
/*  E=(A : Tail), agentAction(A),  */
              Pol=nil, reward(V,S), (Prob is 1.0) .


  /* ---- Some useful predicates mentioned in the interpreter ----- */

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.


deterministic(A,S) :- not nondetActions(A,S,OutcomesList).


range([D | Tail],[D | Tail]).     % Tail can be []

/* 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(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)).
```

C.2 An Example: Coins

```
/*                  Coins           */

/* The predicate range(F,List) defines the finite range of values.
   This predicate occurs in the definition of finite version of "pi".
*/
range(coins2,[1,3]).
range(coins5,[1,2,3,4,5]).
range(oddCoins,[1,3,5]).
range(evenCoins,[2,4]).


/* Call
                   bp(prog1, 1, Pol, U, Prob, outputFile).
                   bp(prog2, 1, Pol, U, Prob, outputFile).
   where prog is a Golog program, 1 is the horizon, Pol is an optimal
   policy, U is its expected utility, Prob is its probability of
   success. The computed policy will be printed into outputFile.

   Programs (prog1) and (prog2) illustrate some tricky interactions
   between nondeterminism in the program and stochastic actions.
   Note that prog1 and prog2 lead to different policies.
*/
proc(prog1,   flip(1) : (?(head(1)) # ?(-head(1))) ).

proc(prog2, (flip(1) : ?(head(1)) ) # (flip(1) : ?(-head(1)) ) ).


/* Call
                   bp(search,5,Pol,U,Prob,outputFile).
                   bp(constr,5,Pol,U,Prob,outputFile).
                   bp(best,5,Pol,U,Prob,outputFile).

 Choose the best policy of flipping given the horizon 5. According to
 the reward function, the optimal policy follows this sequence:
               coin 1, coin 3, coin 5, coin 2, coin 4.
 The programs 'search' and 'best' do not provide any constraints, but
 the program 'constr' indicates that coins 2 and 4 must be attempted
 only after coins 1,3,5. Given the program 'constr', the computation of
 an optimal policy takes 1 sec, and given the program 'search'
 (the program 'best', respectively), the computation takes 27sec (39sec),
 on computer with two 300Mhz processors and 128Mb of RAM.

Calls
                   bp(search,6,Pol,U,Prob,outputFile).
                   bp(search,7,Pol,U,Prob,outputFile).
```

```
compute policies for longer horizons.
*/

proc(search,
        (flip(1) # flip(2) # flip(3) # flip(4) # flip(5)) : search
).

proc(best,
           pickBest(acoin,coins5,flip(acoin)) : best
).

proc(constr,
         if( head(1) & head(3) & head(5),
                  (flip(2) # flip(4)) : constr,
                      (flip(1) # flip(3) # flip(5)) : constr)
).

proc(odd,
           pickBest(acoin,oddCoins,flip(acoin)) : odd
).

/* Stochastic actions have a finite number of outcomes:
   we list all of them
*/

nondetActions(flip(Coin),S,[flipHead(Coin),flipTail(Coin)]).

/* Using predicate  prob(Outcome,Probability,Situation)
   we specify numerical values of probabilities for each outcome
*/

prob(flipHead(X), 0.5, S).  prob(flipTail(X), 0.5, S).

/* We formulate precondition axioms using the predicate
poss(Outcome, Situation)
The right-hand side of precondition axioms provides conditions
under which Outcome is possible in Situation
*/

poss(flipHead(X),S).  poss(flipTail(X),S).

/* head(C,S) is true if the coin C is heads up in S  */

head(C,do(A,S)) :- A=flipHead(C) ;
               head(C,S),  A \= flipTail(C).
```

```
reward(0,s0).
reward(R,do(A,S)) :- A=flipTail(X), R is 0.

reward(R,do(A,S)) :- A = flipHead(1),
( head(C,S), R is -10 ; not head(C,S), R is 100 ).

reward(R,do(A,S)) :- A = flipHead(3),
  (head(1,S), not head(2,S), not head(3,S), not head(4,S), not head(5,S),
              R is 300 ; R is -30 ).

reward(R,do(A,S)) :- A = flipHead(5),
  (head(1,S), head(3,S), not head(2,S), not head(4,S), not head(5,S),
                         R is 500 ; R is -50 ).

reward(R,do(A,S)) :- A = flipHead(2),
  (head(1,S), head(3,S), head(5,S), not head(2,S), not head(4,S),
                                    R is 200 ; R is -20 ).

reward(R,do(A,S)) :- A = flipHead(4),
(head(1,S), head(3,S), head(5,S), head(2,S), not head(4,S),
                                                    R is 400 ; R is -40 ).


/*  The predicate  senseCondition(Outcome,Psi) describes what logical
    formula Psi should be evaluated to determine Outcome uniquely
*/

senseCondition(flipHead(X),head(X)).
senseCondition(flipTail(X),(-head(X))).

/* Agent actions vs nature's actions: the former are those which can be
  executed by agents, the latter (outcomes) can be executed only by nature
*/

agentAction(flip(C)).

restoreSitArg(head(Coin),S,head(Coin,S)).
```

C.3 An Optimal Policy for Tossing Coins

```
This report is started at time Thu Apr  4 18:02:13 2002

The Golog program is
 proc(search,
  (flip(1) # flip(2) # flip(3) # flip(4) # flip(5)) : search)
The call
    bp(search,6,Pol,U,Prob,outputFile).
leads to the following computation.

Time elapsed is 383 seconds
The optimal policy is
flip(1) : senseEffect(flip(1)) :
  if(head(1), flip(3) : senseEffect(flip(3)) :
    if(head(3), flip(5) : senseEffect(flip(5)) :
      if(head(5), flip(2) : senseEffect(flip(2)) :
        if(head(2), flip(4) : senseEffect(flip(4)) :
          if(head(4), flip(1) : senseEffect(flip(1)) :
            if(head(1), nil, ?(-(head(1))) : nil),
          ?(-(head(4))) : flip(4) : senseEffect(flip(4)) :
             if(head(4), nil, ?(-(head(4))) : nil)),
        ?(-(head(2))) : flip(2) : senseEffect(flip(2)) :
          if(head(2), flip(4) : senseEffect(flip(4)) :
            if(head(4), nil, ?(-(head(4))) : nil),
          ?(-(head(2))) : flip(2) : senseEffect(flip(2)) :
            if(head(2), nil, ?(-(head(2))) : nil))),
      ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
        if(head(5), flip(2) : senseEffect(flip(2)) :
          if(head(2), flip(4) : senseEffect(flip(4)) :
            if(head(4), nil, ?(-(head(4))) : nil),
          ?(-(head(2))) : flip(2) : senseEffect(flip(2)) :
            if(head(2), nil, ?(-(head(2))) : nil)),
        ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
          if(head(5), flip(2) : senseEffect(flip(2)) :
            if(head(2), nil, ?(-(head(2))) : nil),
          ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
            if(head(5), nil, ?(-(head(5))) : nil)))),
    ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
      if(head(3), flip(5) : senseEffect(flip(5)) :
        if(head(5), flip(2) : senseEffect(flip(2)) :
          if(head(2), flip(4) : senseEffect(flip(4)) :
            if(head(4), nil, ?(-(head(4))) : nil),
          ?(-(head(2))) : flip(2) : senseEffect(flip(2)) :
            if(head(2), nil, ?(-(head(2))) : nil)),
        ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
          if(head(5), flip(2) : senseEffect(flip(2)) :
            if(head(2), nil, ?(-(head(2))) : nil),
```

```
        ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
          if(head(5), nil, ?(-(head(5))) : nil))),
    ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
      if(head(3), flip(5) : senseEffect(flip(5)) :
        if(head(5), flip(2) : senseEffect(flip(2)) :
          if(head(2), nil, ?(-(head(2))) : nil),
        ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
          if(head(5), nil, ?(-(head(5))) : nil)),
      ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
        if(head(3), flip(5) : senseEffect(flip(5)) :
          if(head(5), nil, ?(-(head(5))) : nil),
            ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
              if(head(3), nil, ?(-(head(3))) : nil))))),
?(-(head(1))) : flip(1) : senseEffect(flip(1)) :
  if(head(1), flip(3) : senseEffect(flip(3)) :
    if(head(3), flip(5) : senseEffect(flip(5)) :
      if(head(5), flip(2) : senseEffect(flip(2)) :
        if(head(2), flip(4) : senseEffect(flip(4)) :
          if(head(4), nil, ?(-(head(4))) : nil),
        ?(-(head(2))) : flip(2) : senseEffect(flip(2)) :
          if(head(2), nil, ?(-(head(2))) : nil)),
      ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
        if(head(5), flip(2) : senseEffect(flip(2)) :
          if(head(2), nil, ?(-(head(2))) : nil),
        ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
          if(head(5), nil, ?(-(head(5))) : nil))),
    ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
      if(head(3), flip(5) : senseEffect(flip(5)) :
        if(head(5), flip(2) : senseEffect(flip(2)) :
          if(head(2), nil, ?(-(head(2))) : nil),
        ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
          if(head(5), nil, ?(-(head(5))) : nil)),
      ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
        if(head(3), flip(5) : senseEffect(flip(5)) :
          if(head(5), nil, ?(-(head(5))) : nil),
            ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
              if(head(3), nil, ?(-(head(3))) : nil)))),
  ?(-(head(1))) : flip(1) : senseEffect(flip(1)) :
    if(head(1), flip(3) : senseEffect(flip(3)) :
      if(head(3), flip(5) : senseEffect(flip(5)) :
        if(head(5), flip(2) : senseEffect(flip(2)) :
          if(head(2), nil, ?(-(head(2))) : nil),
        ?(-(head(5))) : flip(5) : senseEffect(flip(5)) :
          if(head(5), nil, ?(-(head(5))) : nil)),
      ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
        if(head(3), flip(5) : senseEffect(flip(5)) :
          if(head(5), nil, ?(-(head(5))) : nil),
```

```
          ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
            if(head(3), nil, ?(-(head(3))) : nil))),
      ?(-(head(1))) : flip(1) : senseEffect(flip(1)) :
        if(head(1), flip(3) : senseEffect(flip(3)) :
          if(head(3), flip(5) : senseEffect(flip(5)) :
            if(head(5), nil, ?(-(head(5))) : nil),
          ?(-(head(3))) : flip(3) : senseEffect(flip(3)) :
            if(head(3), nil, ?(-(head(3))) : nil)),
        ?(-(head(1))) : flip(1) : senseEffect(flip(1)) :
          if(head(1), flip(3) : senseEffect(flip(3)) :
            if(head(3), nil, ?(-(head(3))) : nil),
          ?(-(head(1))) : flip(1) : senseEffect(flip(1)) :
            if(head(1), nil, ?(-(head(1))) : nil))))))
The value of the optimal policy is 806.094
The probability of successful termination is 1.0
```

C.4 ADD-based Representation of a Delivery Example

```
Fluents:
hcX - a person X has coffee
hmX - a person X has mail
rl - the robot location
g  - going /* not considered, we can abstract it away */
cf - the robot is carrying coffee
cmX - the robot is carrying mail for a person X
wcX - wantsCoffee (one of people)
mpX - mailPresent (for one of people)
Actions:
pc - pickup coffee
pmAn - pickup mail for Ann
pmJo - pickup mail for Joe
gcAn - give coffee to Ann
gcJo - give coffee to Joe
gmAn - give mail to Ann
gmJo - give mail to Joe
gotoMO - go to the main office from the current location
gotoAn - go to Ann's office from the current location
gotoJo - go to Joe's office from the current location
pbc  - put back coffee
pbmAn - put Ann's mail back into his mailbox
pbmJo - put Joe's mail back into his mailbox
--------------------------------------------------------------------
(variables (hcAn yes no) (hcJo yes no) (hmAn yes no) (hmJo yes no)
(rl mo hl an jo) (cf yes no) (cmAn yes no) (cmJo yes no)
(wcAn yes no) (wcJo yes no) (mpAn yes no) (mpJo yes no))
action pc
   hcAn   (hcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hcJo   (hcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmAn   (hmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmJo   (hmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   rl     (rl     (mo    (1.0  0.0  0.0  0.0))
                  (hl    (0.0  1.0  0.0  0.0))
                  (an    (0.0  0.0  1.0  0.0))
                  (jo    (0.0  0.0  0.0  1.0)))
   cf     (cf     (yes   (1.0 0.0))
                  (no    (rl   (mo   (1.0 0.0))
                               (hl   (0.0 1.0))
                               (an   (0.0 1.0))
                               (jo   (0.0 1.0)))))
```

```
   cmAn   (cmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   cmJo   (cmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcAn   (wcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcJo   (wcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   mpAn   (mpAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   mpJo   (mpJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
endaction
action pmAn
   hcAn   (hcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hcJo   (hcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmAn   (hmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmJo   (hmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   rl     (rl     (mo    (1.0  0.0  0.0  0.0))
                  (hl    (0.0  1.0  0.0  0.0))
                  (an    (0.0  0.0  1.0  0.0))
                  (jo    (0.0  0.0  0.0  1.0)))
   cf     (cf     (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   cmAn   (cmAn   (yes   (1.0 0.0))
                  (no    (mpAn   (yes   (rl   (mo   (1.0 0.0))
                                              (hl   (0.0 1.0))
                                              (an   (0.0 1.0))
                                              (jo   (0.0 1.0))))
                                 (no   (0.0  1.0)))))
   cmJo   (cmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcAn   (wcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcJo   (wcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   mpAn   (mpAn   (yes   (0.0 1.0))
                  (no    (0.0 1.0)))
   mpJo   (mpJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
endaction
action pmJo
   hcAn   (hcAn   (yes   (1.0 0.0))
```

```
                  (no     (0.0 1.0)))
   hcJo   (hcJo   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   hmAn   (hmAn   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   hmJo   (hmJo   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   rl     (rl     (mo     (1.0  0.0  0.0  0.0))
                  (hl     (0.0  1.0  0.0  0.0))
                  (an     (0.0  0.0  1.0  0.0))
                  (jo     (0.0  0.0  0.0  1.0)))
   cf     (cf     (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   cmAn   (cmAn   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   cmJo   (cmJo   (yes    (1.0 0.0))
                  (no     (mpJo   (yes   (rl   (mo   (1.0 0.0))
                                               (hl   (0.0 1.0))
                                               (an   (0.0 1.0))
                                               (jo   (0.0 1.0))))
                                  (no   (0.0  1.0)))))
   wcAn   (wcAn   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   wcJo   (wcJo   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   mpAn   (mpAn   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   mpJo   (mpJo   (yes    (0.0 1.0))
                  (no     (0.0 1.0)))
endaction
action gcAn
   hcAn   (hcAn   (yes    (1.0 0.0))
                  (no     (cf   (yes   (rl   (mo   (0.0  1.0))
                                             (hl   (0.0  1.0))
                                             (an   (0.95  0.05))
                                             (jo   (0.0  1.0))))
                                (no   (0.0  1.0)))))
   hcJo   (hcJo   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   hmAn   (hmAn   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   hmJo   (hmJo   (yes    (1.0 0.0))
                  (no     (0.0 1.0)))
   rl     (rl     (mo     (1.0  0.0  0.0  0.0))
                  (hl     (0.0  1.0  0.0  0.0))
                  (an     (0.0  0.0  1.0  0.0))
                  (jo     (0.0  0.0  0.0  1.0)))
```

```
   cf      (cf      (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   cmAn    (cmAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   cmJo    (cmJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   wcAn    (wcAn    (yes    (cf    (yes    (rl    (mo    (1.0   0.0))
                                                  (hl    (1.0   0.0))
                                                  (an    (0.05   0.95))
                                                  (jo    (1.0   0.0))))
                                   (no    (1.0   0.0))))
                    (no     (0.0 1.0)))
   wcJo    (wcJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   mpAn    (mpAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   mpJo    (mpJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
endaction
action gcJo
   hcAn    (hcAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   hcJo    (hcJo    (yes    (1.0 0.0))
                    (no     (cf    (yes    (rl    (mo    (0.0   1.0))
                                                  (hl    (0.0   1.0))
                                                  (an    (0.0   1.0))
                                                  (jo    (0.95   0.05))))
                                   (no    (0.0   1.0)))))
   hmAn    (hmAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   hmJo    (hmJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   rl      (rl      (mo     (1.0   0.0   0.0   0.0))
                    (hl     (0.0   1.0   0.0   0.0))
                    (an     (0.0   0.0   1.0   0.0))
                    (jo     (0.0   0.0   0.0   1.0)))
   cf      (cf      (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   cmAn    (cmAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   cmJo    (cmJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   wcAn    (wcAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   wcJo    (wcJo    (yes    (cf    (yes    (rl    (mo    (1.0   0.0))
                                                  (hl    (1.0   0.0))
                                                  (an    (1.0   0.0))
```

```
                                                   (jo    (0.05   0.95))))
                                       (no    (1.0   0.0))))
                          (no     (0.0 1.0)))
   mpAn      (mpAn        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   mpJo      (mpJo        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
endaction
action gmAn
   hcAn      (hcAn        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   hcJo      (hcJo        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   hmAn      (hmAn        (yes    (1.0 0.0))
                          (no     (cmAn    (yes    (rl    (mo    (0.0   1.0))
                                                          (hl    (0.0   1.0))
                                                          (an    (0.95  0.05))
                                                          (jo    (0.0   1.0))))
                                           (no    (0.0   1.0)))))
   hmJo      (hmJo        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   rl        (rl          (mo     (1.0  0.0  0.0  0.0))
                          (hl     (0.0  1.0  0.0  0.0))
                          (an     (0.0  0.0  1.0  0.0))
                          (jo     (0.0  0.0  0.0  1.0)))
   cf        (cf          (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   cmAn      (cmAn        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   cmJo      (cmJo        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   wcAn      (wcAn        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   wcJo      (wcJo        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   mpAn      (mpAn        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   mpJo      (mpJo        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
endaction
action gmJo
   hcAn      (hcAn        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   hcJo      (hcJo        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
   hmAn      (hmAn        (yes    (1.0 0.0))
                          (no     (0.0 1.0)))
```

```
   hmJo   (hmJo   (yes   (1.0 0.0))
                  (no    (cmJo   (yes   (rl   (mo   (0.0  1.0))
                                              (hl   (0.0  1.0))
                                              (an   (0.0  1.0))
                                              (jo   (0.95  0.05))))
                                 (no   (0.0  1.0)))))
   rl     (rl     (mo    (1.0  0.0  0.0  0.0))
                  (hl    (0.0  1.0  0.0  0.0))
                  (an    (0.0  0.0  1.0  0.0))
                  (jo    (0.0  0.0  0.0  1.0)))
   cf     (cf     (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   cmAn   (cmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   cmJo   (cmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcAn   (wcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcJo   (wcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   mpAn   (mpAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   mpJo   (mpJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
endaction
action gotoMO
   hcAn   (hcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hcJo   (hcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmAn   (hmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmJo   (hmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   rl     (rl     (mo    (0.99  0.01  0.0  0.0))
                  (hl    (0.99  0.01  0.0  0.0))
                  (an    (0.99  0.01  0.0  0.0))
                  (jo    (0.99  0.01  0.0  0.0)))
   cf     (cf     (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   cmAn   (cmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   cmJo   (cmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcAn   (wcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcJo   (wcJo   (yes   (1.0 0.0))
```

```
                  (no    (0.0 1.0)))
   mpAn   (mpAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   mpJo   (mpJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
endaction
action gotoAn
   hcAn   (hcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hcJo   (hcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmAn   (hmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmJo   (hmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   rl     (rl     (mo    (0.0  0.01  0.99  0.0))
                  (hl    (0.0  0.01  0.99  0.0))
                  (an    (0.0  0.01  0.99  0.0))
                  (jo    (0.0  0.01  0.99  0.0)))
   cf     (cf     (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   cmAn   (cmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   cmJo   (cmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcAn   (wcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcJo   (wcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   mpAn   (mpAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   mpJo   (mpJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
endaction
action gotoJo
   hcAn   (hcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hcJo   (hcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmAn   (hmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmJo   (hmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   rl     (rl     (mo    (0.0  0.01  0.0  0.99))
                  (hl    (0.0  0.01  0.0  0.99))
                  (an    (0.0  0.01  0.0  0.99))
                  (jo    (0.0  0.01  0.0  0.99)))
```

```
   cf      (cf      (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   cmAn    (cmAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   cmJo    (cmJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   wcAn    (wcAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   wcJo    (wcJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   mpAn    (mpAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   mpJo    (mpJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
endaction
action pbc
   hcAn    (hcAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   hcJo    (hcJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   hmAn    (hmAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   hmJo    (hmJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   rl      (rl      (mo     (1.0  0.0  0.0  0.0))
                    (hl     (0.0  1.0  0.0  0.0))
                    (an     (0.0  0.0  1.0  0.0))
                    (jo     (0.0  0.0  0.0  1.0)))
   cf      (cf      (yes    (rl    (mo    (0.0 1.0))
                                   (hl    (1.0 0.0))
                                   (an    (1.0 0.0))
                                   (jo    (1.0 0.0))))
                    (no     (0.0 1.0)))
   cmAn    (cmAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   cmJo    (cmJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   wcAn    (wcAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   wcJo    (wcJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   mpAn    (mpAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   mpJo    (mpJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
endaction
action pbmAn
```

```
   hcAn   (hcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hcJo   (hcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmAn   (hmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmJo   (hmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   rl     (rl     (mo    (1.0  0.0  0.0  0.0))
                  (hl    (0.0  1.0  0.0  0.0))
                  (an    (0.0  0.0  1.0  0.0))
                  (jo    (0.0  0.0  0.0  1.0)))
   cf     (cf     (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   cmAn   (cmAn   (yes   (rl   (mo   (0.0 1.0))
                               (hl   (1.0 0.0))
                               (an   (1.0 0.0))
                               (jo   (1.0 0.0))))
                  (no    (0.0 1.0)))
   cmJo   (cmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcAn   (wcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   wcJo   (wcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   mpAn   (mpAn   (yes   (1.0 0.0))
                  (no    (rl   (mo   (1.0 0.0))
                               (hl   (0.0 1.0))
                               (an   (0.0 1.0))
                               (jo   (0.0 1.0)))))
   mpJo   (mpJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
endaction
action pbmJo
   hcAn   (hcAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hcJo   (hcJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmAn   (hmAn   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   hmJo   (hmJo   (yes   (1.0 0.0))
                  (no    (0.0 1.0)))
   rl     (rl     (mo    (1.0  0.0  0.0  0.0))
                  (hl    (0.0  1.0  0.0  0.0))
                  (an    (0.0  0.0  1.0  0.0))
                  (jo    (0.0  0.0  0.0  1.0)))
   cf     (cf     (yes   (1.0 0.0))
```

```
                    (no     (0.0 1.0)))
   cmAn    (cmAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   cmJo    (cmJo    (yes    (rl    (mo    (0.0 1.0))
                                   (hl    (1.0 0.0))
                                   (an    (1.0 0.0))
                                   (jo    (1.0 0.0))))
                    (no     (0.0 1.0)))
   wcAn    (wcAn    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   wcJo    (wcJo    (yes    (1.0 0.0))
                    (no     (0.0 1.0)))
   mpAn    (mpAn    (yes    (1.0 0.0))
                    (no     (1.0 0.0)))
   mpJo    (mpJo    (yes    (1.0 0.0))
                    (no     (rl    (mo    (1.0 0.0))
                                   (hl    (0.0 1.0))
                                   (an    (0.0 1.0))
                                   (jo    (0.0 1.0)))))
endaction
reward  (hcAn  (yes  (hmAn  (yes  (hcJo  (yes  (hmJo  (yes  (55.0))
                                                       (no   (25.0))))
                                         (no  (hmJo   (yes  (55.0))
                                                   (no   (55.0))))))
                            (no   (hcJo  (yes  (hmJo  (yes  (55.0))
                                                       (no   (25.0))))
                                         (no   (hmJo  (yes  (55.0))
                                         (no   (25.0))))))))
                 (no  (hmAn  (yes  (hcJo  (yes  (hmJo  (yes  (30.0))
                                                       (no   (0.0))))
                                         (no  (hmJo   (yes  (30.0))
                                                   (no   (30.0))))))
                            (no   (hcJo  (yes  (hmJo  (yes  (30.0))
                                                       (no   (0.0))))
                                         (no   (hmJo  (yes  (30.0))
                                                   (no   (0.0)))))))))
discount 1.000000
horizon 5.000000
```

C.5 A Situation Calculus Representation of a FACTORY Example

```
:- set_flag(all_dynamic, on).
:- dynamic(proc/2).
:- dynamic(skilledLabourPresent/1).
:- dynamic(hasSprayGun/1).
:- dynamic(hasGlue/1).
:- dynamic(hasBolts/1).
:- pragma(debug).

/* This file is  "myfactory6"; version 6 of SPUDD test factory examples.
35 FLUENTS (grounded)

% The first 17 variables are mentioned in factory.dat example
t : Type_Needed       A high (true) or low (false) quality job is required
c : Connected         Objects a and b are connected
cw : Connected_Well   Objects a and b are well connnected
ap : A_painted        Object a is painted
apw : A_painted_well  Object a is well painted
bp : B_painted        Object b is painted
bpw : B_painted       Object b is well painted
ash : A_shaped        Object a is shaped
bsh : B_shaped        Object b is shaped
asm : A_smoothed      Object a is smoothed
bsm : B_smoothed      Object b is smoothed
adr : A_drilled       Object a is drilled
bdr : B_drilled       Object b is drilled
bo : Bolts            The robot has bolts
gl : Glue             The robot has glue
sg : Spray_Gun        The robot has a spray gun
sl : Skilled_Labour   There is skilled labour present

% The following 2 variables are mentioned in factory0.dat and
% all subsequent examples
cl : Clamps            The robot has clamps
dr : Drill             The robot has a drill

% The following 2 variables are mentioned in factory1.dat and
% all subsequent examples. Thus, factory1.dat has 21 boolean variables
mo : Mounting           The robot has a mounting device
br : Brushes            The robot has brushes

% The following 7 variables are mentioned in factory4.dat and
% all subsequent examples. Thus, factory4.dat has 28 boolean variables
la : Laquer              Laquer is available
aw : arc-welder          The robot has an arc-welder
```

```
sw : spot welder         The robot has a spot-welder
swl : arc-welder labour  Skilled labour for the arc-welder is available
bit : bit                The robot has bits for the drill
awe : arc-welds          The robot has welding materials for the arc-welder
swe : spot-welds         The robot has welding materials for the spot-welder

% The following 7 variables are mentioned in factory6.dat example. The first
% two variables "cac" and "cbc" occur in factory5.dat and all
% remaining 5 variables occur only in the last example factory6.dat
% Thus, factory6.dat has 35 boolean variables and the size of
% the state space is 2^35 (i.e., approximately 3.4x10^10) states.
cac : CA_Connected        Objects C and A are connected
cbc : CB_Connected        Objects C and B are connected
cp : C_painted            Object c is painted
cpw : C_painted_well      Object c is well painted
csh : C_shaped            Object c is shaped
csm : C_smoothed          Object c is smoothed
cdr : C_drilled           Object c is drilled

14 ACTIONS (grounded)
% These 14 actions are used in all factory related examples
Shape object a
Shape object b
Drill object a
Drill object b
Dip (paint) object a
Dip (paint) object b
Spray (paint) object a
Spray (paint) object b
Hand-paint object a
Hand-paint object b
Bolt two objects A and B together
Glue two objects together
Polish object a
Polish object b

% The following action is used in all examples starting from factory3.dat
Weld objects A and B

% factory6.dat example has additional 8 actions (23 actions in total)
% The first two actions occur in factory5.dat example and all remaining
% 7 actions are allowed only in the factory6.dat example.
% Because the situation calculus representation is using action terms
% weld(x,y) and glue(x,y), where the variables x and y vary over objects
% {a,b,c} allowed in the factory domain, the axiomatization allows
% several additional actions (e.g., weld(a,c), weld(b,c), glue(a,c),
% glue(b,c)) that are not  allowed in the factory6.dat example.
```

```
Bolt objects A and C together
Bolt objects B and C together
Shape object c
Drill object c
Dip (paint) object c
Spray (paint) object c
Hand-paint object c
Polish object c
*/
/* Nondeterministic actions */

/* The action "spray(X)" might have four different outcomes:
   a part becomes painted and well painted - nature's action is "spraySW",
a part becomes painted but not well painted - nature's action is "spraySnW",
a part remains unpainted, but is well painted - nature's action is "sprayFW",
a part remains unpainted and not well painted - nature's action is "sprayFnW".
   According to the probabilities of transitions between states choosen for
   the factory0 example in the SPUDD demo version, the probability of becoming
 painted is 0.9 (a part remains unpainted with 0.1 probability). The probability
   of becoming well painted is 0.44 (a part will not be well painted with
   the probability 0.56)
*/
nondetActions(spray(X), S, [spraySW(X), spraySnW(X), sprayFW(X), sprayFnW(X)]).

nondetActions(shape(X), S, [shapeS(X), shapeF(X)]).

nondetActions(handPaint(X), S, [handPaintS(X), handPaintF(X)]).

nondetActions(polish(X), S, [polishS(X), polishF(X)]).

nondetActions(drill(X), S, [drillS(X), drillF(X)]).

nondetActions(weld(X,Y), S, [weldS(X,Y), weldF(X,Y)]).


        /* Identification conditions for stochastic actions */

senseCondition(spraySW(Obj), Phi) :- Phi=( painted(Obj) & paintedWell(Obj) ).
senseCondition(spraySnW(Obj), Phi) :- Phi=( painted(Obj) & (-paintedWell(Obj)) )
senseCondition(sprayFW(Obj), Phi) :- Phi=( (-painted(Obj)) & paintedWell(Obj) ).
senseCondition(sprayFnW(Obj),Phi) :- Phi=((-painted(Obj)) & (-paintedWell(Obj)))

senseCondition(shapeS(Obj), Phi) :- Phi=shaped(Obj) .
senseCondition(shapeF(Obj), Phi) :- Phi=(-shaped(Obj)) .

senseCondition(handPaintS(X), Phi) :-  Phi=paintedWell(X).
senseCondition(handPaintF(X), Phi) :-  Phi=(-paintedWell(X)).
```

```
senseCondition(polishS(Obj), Phi) :- Phi=smoothed(Obj) .
senseCondition(polishF(Obj), Phi) :- Phi=(-smoothed(Obj)) .

senseCondition(drillS(Obj), Phi) :- Phi=drilled(Obj) .
senseCondition(drillF(Obj), Phi) :- Phi=(-drilled(Obj)) .

senseCondition(weldS(Obj1,Obj2), Phi) :- Phi=connectedWell(Obj1,Obj2) .
senseCondition(weldF(Obj1,Obj2), Phi) :- Phi=(-connectedWell(Obj1,Obj2)) .


                          /* Probabilities */

prob(spraySW(X),Pr,S) :-  hasMountingDevice(S), ( Pr is 0.9*0.44 ) .
prob(spraySnW(X),Pr,S) :- hasMountingDevice(S), ( Pr is 0.9*0.56 ) .
prob(sprayFW(X),Pr,S) :-  hasMountingDevice(S), ( Pr is 0.1*0.44 ) .
prob(sprayFnW(X),Pr,S) :- hasMountingDevice(S), ( Pr is 0.1*0.56 ) .

prob(spraySW(X),0,S) :-  not hasMountingDevice(S).
prob(spraySnW(X),1,S) :- not hasMountingDevice(S).
prob(sprayFW(X),0,S) :-  not hasMountingDevice(S).
prob(sprayFnW(X),0,S) :- not hasMountingDevice(S).

prob(shapeS(X),Pr,S) :-  ( Pr is 0.8 ) .
prob(shapeF(X),PrF,S) :- prob(shapeS(X),PrS, S),  (PrF is 1.0 - PrS ) .

prob(handPaintS(X),Pr,S) :-  ( Pr is 0.8 ) .
prob(handPaintF(X),PrF,S) :- prob(handPaintS(X),PrS, S),  (PrF is 1.0 - PrS ) .

prob(polishS(X),Pr,S) :-  ( Pr is 0.8 ) .
prob(polishF(X),PrF,S) :-  prob(polishS(X),PrS, S),  (PrF is 1.0 - PrS ) .

prob(drillS(X),Pr,S) :-  ( Pr is 0.8 ) .
prob(drillF(X),PrF,S) :-  prob(drillS(X),PrS, S),  (PrF is 1.0 - PrS ) .

prob(weldS(X,Y),Pr,S) :- ( Pr is 0.9 ) .
prob(weldF(X,Y),PrF,S) :- prob(weldS(X,Y),PrS, S), (PrF is 1.0 - PrS ) .


                 /* Precondition axioms */

poss(spraySW(X),S) :- object(X), hasSprayGun(S).
poss(spraySnW(X),S) :- object(X), hasSprayGun(S).
poss(sprayFW(X),S) :- object(X), hasSprayGun(S).
poss(sprayFnW(X),S) :- object(X), hasSprayGun(S).

poss(shapeS(X),S) :- object(X), not (object(Y), not X=Y, connected(X,Y,S)),
```

```
                    not (object(Y), not X=Y, connectedWell(X,Y,S)).

poss(shapeF(X),S) :- object(X), not (object(Y), not X=Y, connected(X,Y,S)),
                    not (object(Y), not X=Y, connectedWell(X,Y,S)).

poss(handPaintS(X),S) :- object(X),  hasBrushes(S).
poss(handPaintF(X),S) :- object(X),  hasBrushes(S).

poss(polishS(X),S) :- object(X).
poss(polishF(X),S) :- object(X).

poss(drillS(X),S) :- object(X).
poss(drillF(X),S) :- object(X).

poss(weldS(X,Y),S) :- object(X), object(Y), hasSkilledWelder(S).
poss(weldF(X,Y),S) :- object(X), object(Y), hasSkilledWelder(S).

poss(dip(X),S) :- object(X).

poss(bolt(X,Y),S) :-  drilled(X,S), drilled(Y,S), not X=Y, hasBolts(S).

poss(glue(X,Y),S) :- object(X), object(Y), not X=Y, hasGlue(S), hasClamps(S).


            /*   Successor state axioms */

typeNeeded(JobQuality,do(A,S)) :- typeNeeded(JobQuality,S).
/* In the initial situation s0, arguments can be typeNeeded(highQuality,s0)
   or  typeNeeded(lowQuality,s0)  */

skilledLabourPresent(do(A,S)) :- skilledLabourPresent(S).
hasSprayGun(do(A,S)) :- hasSprayGun(S).
hasGlue(do(A,S)) :- hasGlue(S).
hasBolts(do(A,S)) :- hasBolts(S).

/* hasClamps(S) and hasDrill(S) are introduced in factory0.dat */
hasClamps(do(A,S)) :- hasClamps(S).
hasDrill(do(A,S)) :- hasDrill(S).

/* hasMountingDevice(S) and hasBrushes(S) are introduced in factory1.dat */
hasMountingDevice(do(A,S)) :- hasMountingDevice(S).
hasBrushes(do(A,S)) :- hasBrushes(S).

/* hasLaquer(S) is the only one new fluent in factory2.dat: this fluent is
   not mentioned in any other s.-s. or precondition axiom */
hasLaquer(do(A,S)) :- hasLaquer(S).
```

```
/*
hasSkilledWelder(S),hasArcWelder(S), hasSpotWelder(S) introduced in factory3.dat
hasArcWelder(do(A,S)) :- hasArcWelder(S).
hasSpotWelder(do(A,S)) :- hasSpotWelder(S).
hasSkilledWelder(do(A,S)) :- hasSkilledWelder(S).

/*hasBits(S), hasArcMaterials(S), hasSpotMaterials(S) introduced in factory4.dat
hasArcMaterials(do(A,S)) :- hasArcMaterials(S).
hasSpotMaterials(do(A,S)) :- hasSpotMaterials(S).
hasBits(do(A,S)) :- hasBits(S).


connected(X,Y,do(A,S)) :- A=bolt(X,Y).
connected(X,Y,do(A,S)) :- A=glue(X,Y).

connected(X,Y,do(A,S)) :- A=bolt(Y,X).
connected(X,Y,do(A,S)) :- A=glue(Y,X).

connected(X,Y,do(A,S)) :-
       (A=weldS(X,Y) ; (A=weldS(Y,X) ; (A=weldF(X,Y) ; A=weldF(Y,X)))) ,
   hasArcWelder(S), hasArcMaterials(S).

connected(X,Y,do(A,S)) :-
       not hasArcWelder(S), hasSpotWelder(S), hasSpotMaterials(S),
       ( A=weldS(X,Y) ; A=weldS(Y,X) ).

connected(X,Y,do(A,S)) :- connected(X,Y,S),
        not A=shapeS(X), not A=shapeF(X), not A=shapeS(Y), not A=shapeF(Y).

connectedWell(X,Y,do(A,S)) :- A=bolt(X,Y).
connectedWell(X,Y,do(A,S)) :- A=bolt(Y,X).

connectedWell(X,Y,do(A,S)) :-
       (A=weldS(X,Y) ; (A=weldS(Y,X) ; (A=weldF(X,Y) ; A=weldF(Y,X)))) ,
   ( connected(X,Y,S) -> hasArcMaterials(S) ;
     hasArcWelder(S), hasArcMaterials(S) ).
/*
connectedWell(X,Y,do(A,S)) :-
       (A=weldS(X,Y) ; (A=weldS(Y,X) ; (A=weldF(X,Y) ; A=weldF(Y,X)))) ,
   hasArcWelder(S), hasArcMaterials(S), not connected(X,Y,S).
*/
connectedWell(X,Y,do(A,S)) :- connectedWell(X,Y,S),
       not A=shapeS(X), not A=shapeF(X), not A=shapeS(Y), not A=shapeF(Y),
       not (A=weldS(X,Y), connected(X,Y,S)),
       not (A=weldF(X,Y), connected(X,Y,S)),
       not (A=weldS(Y,X), connected(X,Y,S)),
       not (A=weldF(Y,X), connected(X,Y,S)).
```

```
painted(X,do(A,S)) :- A=dip(X), smoothed(X,S).
painted(X,do(A,S)) :- A=spraySW(X), smoothed(X,S).
painted(X,do(A,S)) :- A=spraySnW(X), smoothed(X,S).
painted(X,do(A,S)) :- A=handPaintS(X), smoothed(X,S).
painted(X,do(A,S)) :- A=handPaintF(X), smoothed(X,S).

painted(X,do(A,S)) :- painted(X,S),
                not A=shapeS(X),  not A=shapeF(X),
                not A=polishS(X),  not A=polishF(X),
                not A=drillS(X),  not A=drillF(X),
                not (object(Y), not X=Y, A=shapeS(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=shapeF(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=drillS(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=drillF(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=polishS(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=polishF(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=dip(Y), connected(X,Y,S)),
not sprayConnected(X,A,S).

sprayConnected(X,A,S) :- object(Y), not X=Y, connected(X,Y,S),
       ( A=spraySW(Y) ; ( A=spraySnW(Y) ; ( A=sprayFW(Y) ; A=sprayFnW(Y)) ) ).

paintedWell(X,do(A,S)) :- A=spraySW(X), smoothed(X,S).
paintedWell(X,do(A,S)) :- A=sprayFW(X), smoothed(X,S).
paintedWell(X,do(A,S)) :- A=handPaintS(X),
                   skilledLabourPresent(S), smoothed(X,S).

paintedWell(X,do(A,S)) :- paintedWell(X,S),
                not A=shapeS(X), not A=shapeF(X),
                not A=polishS(X),  not A=polishF(X),
                not A=drillS(X),  not A=drillF(X),
                not A=dip(X),
                not (object(Y), not X=Y, A=shapeS(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=shapeF(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=drillS(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=drillF(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=polishS(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=polishF(Y), connected(X,Y,S)),
                not (object(Y), not X=Y, A=dip(Y), connected(X,Y,S) ),
                not sprayConnected(X,A,S).

shaped(X,do(A,S)) :- A=shapeS(X), not (object(Y), not X=Y, connected(X,Y,S)).

shaped(X,do(A,S)) :- shaped(X,S),
        not (object(Y), not X=Y, A=drillS(X), connected(X,Y,S)),
        not (object(Y), not X=Y, A=drillF(X), connected(X,Y,S)),
```

```
        not (object(Y), not X=Y, A=drillS(Y), connected(X,Y,S)),
        not (object(Y), not X=Y, A=drillF(Y), connected(X,Y,S)),
        not (object(Y), not X=Y, A=shapeS(X), connected(X,Y,S)),
        not (object(Y), not X=Y, A=shapeF(X), connected(X,Y,S)),
        not (object(Y), not X=Y, A=shapeS(Y), connected(X,Y,S)),
        not (object(Y), not X=Y, A=shapeF(Y), connected(X,Y,S)).

smoothed(X,do(A,S)) :- A=polishS(X), shaped(X,S).

smoothed(X,do(A,S)) :- smoothed(X,S), not A=shapeS(X), not A=shapeF(X),
          not A=drillS(X),  not A=drillF(X),
          not (object(Y), not X=Y, A=shapeS(Y), connected(X,Y,S)),
  not (object(Y), not X=Y, A=shapeF(Y), connected(X,Y,S)),
  not (object(Y), not X=Y, A=drillS(Y), connected(X,Y,S)),
  not (object(Y), not X=Y, A=drillF(Y), connected(X,Y,S)).

drilled(X,do(A,S)) :- A=drillS(X), hasDrill(S), hasBits(S),
          not (object(Y), not X=Y, connected(X,Y,S)).

drilled(X,do(A,S)) :- drilled(X,S),
   not (object(Y), not X=Y, A=drillS(X), connected(X,Y,S)),
   not (object(Y), not X=Y, A=drillF(X), connected(X,Y,S)),
   not (object(Y), not X=Y, A=drillS(Y), connected(X,Y,S)),
   not (object(Y), not X=Y, A=drillF(Y), connected(X,Y,S)).


/* Reward function */

reward(R, s0) :- R is 0.

/* high quality */

reward(R,S) :- typeNeeded(highQuality,S),
               not connected(a,b,S),  R is 0.

reward(R,S) :- typeNeeded(highQuality,S),
               connected(a,b,S), not connected(a,c,S), R is 0.

reward(R,S) :- typeNeeded(highQuality,S),
               connected(a,b,S), connected(a,c,S),
               not (connected(b,c,S), connectedWell(a,b,S)),
               R is 0.

reward(R,S) :- typeNeeded(highQuality,S),
              connected(a,b,S), connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              (not painted(a,S), R is 1 ;
```

```
               painted(a,S),
                  ( not paintedWell(a,S), R is 1 ;
                    paintedWell(a,S),
                       ( not painted(b,S), R is 2 ;
                         painted(b,S),
                            ( not paintedWell(b,S), R is 4 ;
                              paintedWell(b,S),
                                 ( not painted(c,S), R is 6 ;
                                   painted(c,S),
                                      (not paintedWell(c,S), R is 8 ;
                                       paintedWell(c,S), R is 10
                                       )
                                 )
                            )
                       )
                  )
               ).
/*
reward(R,S) :- typeNeeded(highQuality,S),
                connected(a,b,S), connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                not painted(a,S), R is 1.

reward(R,S) :- typeNeeded(highQuality,S),
                connected(a,b,S), connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), not paintedWell(a,S), R is 1.

reward(R,S) :- typeNeeded(highQuality,S),
                connected(a,b,S), connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),
                not painted(b,S), R is 2.

reward(R,S) :- typeNeeded(highQuality,S),
                connected(a,b,S), connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),
                painted(b,S), not paintedWell(b,S),  R is 4.

reward(R,S) :- typeNeeded(highQuality,S),
                connected(a,b,S), connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),
                painted(b,S), paintedWell(b,S), not painted(c,S), R is 6.

reward(R,S) :- typeNeeded(highQuality,S),
```

```
                connected(a,b,S), connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),
                painted(b,S), paintedWell(b,S),
                painted(c,S), not paintedWell(c,S), R is 8.

reward(R,S) :- typeNeeded(highQuality,S),
                connected(a,b,S), connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),
                painted(b,S), paintedWell(b,S),
                painted(c,S), paintedWell(c,S), R is 10.
*/
/* low quality */
reward(R,S) :- typeNeeded(lowQuality,S),
                not connected(a,b,S), R is 0.

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), not connected(a,c,S), R is 0.

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), connected(a,c,S),
                not connected(b,c,S), R is 0.

%-----------------------------------------------------------
reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), not connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
   not painted(a,S),          % Part "a" is not painted
   not painted(b,S), R is 2.

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), not connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
   not painted(a,S),          % Part "a" is not painted
   painted(b,S), not paintedWell(b,S), R is 3.

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), not connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
   not painted(a,S),          % Part "a" is not painted
                painted(b,S), paintedWell(b,S),        % "b" is painted and well
                not painted(c,S), R is 3.

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), not connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
```

```
   not painted(a,S),          % Part "a" is not painted
              painted(b,S), paintedWell(b,S),        % "b" is painted and well
              painted(c,S), not paintedWell(c,S), R is 4.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), not connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
   not painted(a,S),          % Part "a" is not painted
              painted(b,S), paintedWell(b,S),        % "b" is painted and well
              painted(c,S), paintedWell(c,S), R is 3.
%----------------------------------------------------------
reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), not connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), not paintedWell(a,S),  %"a" is painted, but not wel
   not painted(b,S), R is 4.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), not connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), not paintedWell(a,S),  %"a" is painted, but not wel
   painted(b,S), not paintedWell(b,S), R is 5.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), not connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), not paintedWell(a,S), % "a" is painted, but not wel
              painted(b,S), paintedWell(b,S),     % "b" is painted and well
              not painted(c,S), R is 4.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), not connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), not paintedWell(a,S), % "a" is painted, but not wel
              painted(b,S), paintedWell(b,S),     % "b" is painted and well
              painted(c,S), not paintedWell(c,S),
              R is 5.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), not connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), not paintedWell(a,S), % "a" is painted, but not wel
              painted(b,S), paintedWell(b,S),     % "b" is painted and well
              painted(c,S), paintedWell(c,S),
              R is 4.
%----------------------------------------------------------
reward(R,S) :- typeNeeded(lowQuality,S),
```

```
                connected(a,b,S), not connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),  % Part "a" is painted and well
                not painted(b,S),                % "b" is not painted
                R is 3.

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), not connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),     % Part "a" is painted and wel
                painted(b,S), not paintedWell(b,S), % "b" is painted, but not wel
                R is 4.

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), not connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),  % Part "a" is painted and well
                painted(b,S), paintedWell(b,S),  % "b" is painted and well
                not painted(c,S), R is 3.        % "c" is not painted

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), not connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),     % Part "a" is painted and wel
                painted(b,S), paintedWell(b,S),     % "b" is painted and well
                painted(c,S), not paintedWell(c,S), % "c" is painted, but not wel
                R is 4.

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), not connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
                painted(a,S), paintedWell(a,S),     % Part "a" is painted and wel
                painted(b,S), paintedWell(b,S),     % "b" is painted and well
                painted(c,S), paintedWell(c,S),     % "c" is painted and well
                R is 3.
%-----------------------------------------------------------


reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
   not painted(a,S),          % Part "a" is not painted
                not painted(b,S), R is 1.

reward(R,S) :- typeNeeded(lowQuality,S),
                connected(a,b,S), connectedWell(a,b,S),
                connected(a,c,S), connected(b,c,S),
```

```
   not painted(a,S),          % Part "a" is not painted
               painted(b,S), not paintedWell(b,S), R is 2.

reward(R,S) :- typeNeeded(lowQuality,S),
               connected(a,b,S), connectedWell(a,b,S),
               connected(a,c,S), connected(b,c,S),
   not painted(a,S),          % Part "a" is not painted
               painted(b,S), paintedWell(b,S),
               not painted(c,S), R is 2.

reward(R,S) :- typeNeeded(lowQuality,S),
               connected(a,b,S), connectedWell(a,b,S),
               connected(a,c,S), connected(b,c,S),
   not painted(a,S),          % Part "a" is not painted
               painted(b,S), paintedWell(b,S),
               painted(c,S), not paintedWell(c,S), R is 3.

reward(R,S) :- typeNeeded(lowQuality,S),
               connected(a,b,S), connectedWell(a,b,S),
               connected(a,c,S), connected(b,c,S),
                           not painted(a,S),          % Part "a" is not painted
               painted(b,S), paintedWell(b,S),
               painted(c,S), paintedWell(c,S), R is 2.

%--------------------------------------------------------------------
reward(R,S) :- typeNeeded(lowQuality,S),
               connected(a,b,S), connectedWell(a,b,S),
               connected(a,c,S), connected(b,c,S),
   painted(a,S), not paintedWell(a,S),    % Part "a" is painted, but not well
               not painted(b,S), R is 2.

reward(R,S) :- typeNeeded(lowQuality,S),
               connected(a,b,S), connectedWell(a,b,S),
               connected(a,c,S), connected(b,c,S),
  painted(a,S), not paintedWell(a,S),     % Part "a" is painted, but not well
               painted(b,S), not paintedWell(b,S), R is 3.

reward(R,S) :- typeNeeded(lowQuality,S),
               connected(a,b,S), connectedWell(a,b,S),
               connected(a,c,S), connected(b,c,S),
  painted(a,S), not paintedWell(a,S),     % Part "a" is painted, but not well
               painted(b,S), paintedWell(b,S),
               not painted(c,S), R is 2.

reward(R,S) :- typeNeeded(lowQuality,S),
               connected(a,b,S), connectedWell(a,b,S),
               connected(a,c,S), connected(b,c,S),
```

```
  painted(a,S), not paintedWell(a,S),     % Part "a" is painted, but not well
              painted(b,S), paintedWell(b,S),
              painted(c,S), not paintedWell(c,S), R is 3.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
  painted(a,S), not paintedWell(a,S),     % Part "a" is painted, but not well
              painted(b,S), paintedWell(b,S),
              painted(c,S), paintedWell(c,S), R is 2.

%-------------------------------------------------------------------

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), paintedWell(a,S),  %Part "a" is painted and well
              not painted(b,S),                %Part "b" is not painted
              R is 1.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), paintedWell(a,S),  %Part "a" is painted and well
          painted(b,S), not paintedWell(b,S),  %Part "b" is painted, but not we
              R is 1.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), paintedWell(a,S),  %Part "a" is painted and well
              painted(b,S), paintedWell(b,S),  %Part "b" is painted and well
              not painted(c,S),                %Part "c" is not painted
              R is 1.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), paintedWell(a,S),  %Part "a" is painted and well
              painted(b,S), paintedWell(b,S),  %Part "b" is painted and well
          painted(c,S), not paintedWell(c,S),  %Part "c" is painted, but not we
              R is 2.

reward(R,S) :- typeNeeded(lowQuality,S),
              connected(a,b,S), connectedWell(a,b,S),
              connected(a,c,S), connected(b,c,S),
              painted(a,S), paintedWell(a,S),  % Part "a" is painted and well
```

```
               painted(b,S), paintedWell(b,S),  % Part "b" is painted and well
               painted(c,S), paintedWell(c,S),  % Part "c" is painted and well
               R is 1.
%--------------------------------------------------------------------

/* Golog procedures */

proc(new,
  while( some(x, object(x) &
             some(y,  object(y) & -(x=y) &
               -(shaped(x) & connectedWell(x,y))) ),
    if(some(ob, object(ob) & -shaped(ob)),   /* THEN shape it */
       pi(ob, ?(object(ob) & -shaped(ob)) : shape(ob)),  /* ELSE connect it */
         pi(x, ?(some(ob, object(x) & object(ob) & -(x=ob) & -connectedWell(x,ob
             pi(y, ?(object(y) & -(x=y) & -connectedWell(x,y)) :
                 ( drill(y) : drill(x) : bolt(x,y)
                  #
                   if(hasSkilledWelder, weld(x,y), glue(x,y))
                  )
                )
              )
         )
       ) :
      while( some(x, object(x) & -paintedWell(x)),
         if( some(x, object(x) & -smoothed(x)),
                  /* then polish it*/
                pi(x, ?(object(x) & -smoothed(x)) : polish(x)),
                  /* else paint it*/
                pi(x, ?(object(x) & -paintedWell(x)) :
                   if( skilledLabourPresent,
                     /* THEN */
                       handPaint(x),
                         /* ELSE */
                       (spray(x) # dip(x) )
                     )
                   )
                )
            )
).


proc(old,
  while( some(x, object(x) &
             some(y,  object(y) & -(x=y) &
               -(shaped(x) & connectedWell(x,y))) ),
      if(some(ob, object(ob) & -shaped(ob)),   /* THEN shape it */
         pi(ob, ?(object(ob) & -shaped(ob)) : shape(ob)),  /* ELSE connect it */
```

```
    pi(x, ?(some(ob, object(x) & object(ob) & -(x=ob) & -connectedWell(x,ob))) :
            pi(y, ?(object(y) & -(x=y) & -connectedWell(x,y)) :
                  ( if(hasBolts & -(drilled(x) & drilled(y)),
                           /* then */
                        if( drilled(x) & -drilled(y), drill(y), drill(x) ),
                           /* else */
                        bolt(x,y)
                        )
                   #
                    if(hasSkilledWelder, weld(x,y), glue(x,y))
                   )
                )
             )
         )
       ) :
     while( some(x, object(x) & -paintedWell(x)),
         if( some(x, object(x) & -smoothed(x)),
                   /* then polish it*/
                pi(x, ?(object(x) & -smoothed(x)) : polish(x)),
                   /* else paint it*/
                pi(x, ?(object(x) & -paintedWell(x)) :
                    if( skilledLabourPresent,
                      /* THEN */
                       handPaint(x),
                          /* ELSE */ (spray(x) # dip(x) )
                     % if( hasSprayGun, spray(x), dip(x) )
                     )
                    )
                )
          )
).

/* Restore situation arguments to fluents. */

restoreSitArg(typeNeeded(JobQuality),S,typeNeeded(JobQuality,S)).
restoreSitArg(connected(X,Y),S,connected(X,Y,S)).
restoreSitArg(connectedWell(X,Y),S,connectedWell(X,Y,S)).
restoreSitArg(painted(X),S,painted(X,S)).
restoreSitArg(paintedWell(X),S,paintedWell(X,S)).
restoreSitArg(shaped(X),S,shaped(X,S)).
restoreSitArg(smoothed(X),S,smoothed(X,S)).
restoreSitArg(drilled(X),S,drilled(X,S)).
restoreSitArg(skilledLabourPresent,S,skilledLabourPresent(S)).
restoreSitArg(hasSprayGun,S,hasSprayGun(S)).
restoreSitArg(hasGlue,S,hasGlue(S)).
restoreSitArg(hasBolts,S,hasBolts(S)).
```

```
restoreSitArg(hasClamps,S,hasClamps(S)).  /* factory0.dat */
restoreSitArg(hasDrill,S,hasDrill(S)).

restoreSitArg(hasMountingDevice,S,hasMountingDevice(S)). /* factory1.dat */
restoreSitArg(hasBrushes,S,hasBrushes(S)).

restoreSitArg(hasLaquer,S,hasLaquer(S)).  /* factory2.dat */

restoreSitArg(hasSkilledWelder,S,hasSkilledWelder(S)).  /* factory3.dat */
restoreSitArg(hasArcWelder,S,hasArcWelder(S)).
restoreSitArg(hasSpotWelder,S,hasSpotWelder(S)).

restoreSitArg(hasArcMaterials,S,hasArcMaterials(S)).    /* factory4.dat */
restoreSitArg(hasSpotMaterials,S,hasSpotMaterials(S)).
restoreSitArg(hasBits,S,hasBits(S)).


                     /* Primitive Action Declarations */

agentAction(spray(Item)).
agentAction(handPaint(Item)).
agentAction(dip(Item)).
agentAction(shape(Item)).
agentAction(drill(Item)).
agentAction(polish(Item)).
agentAction(glue(Obj1,Obj2)).
agentAction(bolt(Obj1,Obj2)).
agentAction(weld(Obj1,Obj2)).

                       /* Initial Situation */

object(a).
object(b).
object(c).

typeNeeded(highQuality,s0).

skilledLabourPresent(s0).

hasSprayGun(s0).

hasGlue(s0).

hasBolts(s0).

/* hasClamps(S) and hasDrill(S) fluents are introduced in factory0.dat */
hasClamps(s0).
```

```
hasDrill(s0).

hasMountingDevice(s0).
hasBrushes(s0).
hasLaquer(s0).

/* hasSkilledWelder(S), hasArcWelder(S), hasSpotWelder(S) are introduced in
   factory3.dat */
hasSkilledWelder(s0).
hasArcWelder(s0).
hasSpotWelder(s0).

/*hasBits(S), hasArcMaterials(S), hasSpotMaterials(S) introduced in factory4.dat
hasArcMaterials(s0).
hasSpotMaterials(s0).
hasBits(s0).
```

C.6 A Delivery Example in Golog

To run a program mentioned in Section 5.6 do the following. First, load the online interpreter. Second, load this file and call the offline interpreter with arguments:

```
 bp(mail1,12,Pol,Val,Prob,myresults).
 bp(mail2,12,Pol,Val,Prob,myresults).
```

Horizons longer than 12 are ok, but horizons less than 12 are too short to complete both deliveries mentioned in the example.

```
/* Note that Golog programs formulated in this file cannot be run
   with an off-line BestDo interpreter because a delivery domain
   is represented here using actions with temporal arguments and
   BestDo works only with actions without temporal argument.
   To run Golog programs given in this file, first, you need
   to load the online interpreter. It supports actions that have
   temporal argument and provides compatibility with an offline
   interpreter. Then load this file and call the offline interpreter
   with the following arguments:
 bp(gologProgram,Horizon,Pol,Val,Prob,myresults).
*/

:- set_flag(all_dynamic, on).
:- dynamic(proc/2).
:- pragma(debug).
:- dynamic(stophere/1).              % useful for debugging
:- set_flag(stophere/1, spy, on).   % type debug to turn debugger on and use "l"
:- set_flag(stophere/1, leash, stop).%to leap from one break-point to another
/*
:- set_flag(reward/2, spy, on).      % useful for debugging of rewards
:- set_flag(reward/2, leash, stop).  % to enable debugging remove comments
*/

stophere(yes).

/*    A Coffee and Mail Delivery Robot Based on Decision Theoretic Golog.    */
/* Common abbreviations:
   wC  -  wantsCoffee
   tT  -  travelTime
   of  -  office
   cr  -  Craig
   fa  -  Fahiem
   lg  -  lounge
   yv  -  Yves
   vi  -  visitor
   st  -  Steven
   ma  -  Maurice
```

```
*/
                  /*   GOLOG Procedures  */

range(people3, [cr,fa,lg]).
range(people4, [cr,fa,ma,lg]).
range(people41, [cr,fa,st,lg]).
range(people5, [cr,fa,st,ma,lg]).
range(people51, [cr,fa,st,ma,yv]).
range(people6, [cr,vi,fa,st,ma,yv]).
range(people7, [cr,vi,lg,fa,st,ma,yv]).
range(people, [ann,bill,joe]).
range(profs,[cr,fa]).
range(long, [itrc,ray,cr,vi,fa,lg,st,ma,yv]).
range(lp271, [st,ma,yv]).
range(lp276, [fa,vi]).
range(lp290b, [cr]).
range(lp269, [lg]).
range(offices3, [lp271,lp276,lp290b]).
range(offices4, [lp271,lp276,lp290b,lp269]).

inRange(P,R) :- range(R,L), member(P,L).

/* For example, call deliver(people) */
proc(deliver(List),
         pickBest(p, List, pi(t, deliverCoffee(p,t)))
            #
         pickBest(p, List, pi(t, deliverMail(p,t)))
).

proc(mc1,
 while( some(person, some(t1, some(t2, some(number,
         (wantsCoffee(person,t1,t2)
                  v mailPresent(person,number)) & -status(person,out) )))),
          (
           pickBest(p, people3,
             ?(some(t1, some(t2, wantsCoffee(p,t1,t2))) & -status(p,out)) :
                                 pi(t, ?(now(t)): deliverCoffee(p,t)) )
           #
           pickBest(p, people3, ?(some(n, mailPresent(p,n)) & -status(p,out)) :
                                 pi(t, ?(now(t)): deliverMail(p,t)) )
          ) : pi(t, ?(now(t)) : goto(mo,t)) : leaveItems
       )
    ).

/* In the following procedure R can be people3, people4, etc */
proc(cof(R),
 while( some(person, inRange(person,R) &
```

```
           some(t1, some(t2, wantsCoffee(person,t1,t2) &
             -status(person,out) ))),
          loc(pickBest(p, R,
               ?(some(t1, some(t2, wantsCoffee(p,t1,t2))) & -status(p,out)) :
                                     pi(t, ?(now(t)): deliverCoffee(p,t)) ) :
               pi(t, ?(now(t)) : goto(mo,t)) : leaveItems
              )
       )
).

/* In the following procedure R can be offices3 or offices4.  */
proc(coffee(R),
    while( some(office, inRange(office, R) &
             some(person, inRange(person,office) &
              some(t1, some(t2, wantsCoffee(person,t1,t2) &
               -status(person,out) )))),
            pickBest(room, R,
                  limit(pickBest(p,room,
                         ?(some(t1, some(t2, wantsCoffee(p,t1,t2))) &
                             -status(p,out)) :
                             pi(t, ?(now(t)): deliverCoffee(p,t))
                                 ) /*endPickBest*/ :
                        /* pi(t, ?(now(t)) : goto(mo,t)) : leaveItems*/
                          pi(t, ?(now(t)) :
                              if( -carrying(coffee),
                                /* THEN */ goto(mo,t),
                                /* ELSE */ nil  % door closed or give failed
                                    /* or "noOp(tt)" - needs time argument */
                                 ) /*endIF*/
                             )
                        ) /*endLimit*/
                     ) /*endPickBest*/
           ) :  if(robotLoc(mo), nil, pi(t,?(now(t)) : goto(mo,t)))
).


proc(deliverTo(P,T),
  if( robotLoc(mo) & -some(item,carrying(item)),
          /* THEN */     pickItems(P,T) : deliverTo(P,T),
          /* ELSE */
      if( some(item,(carrying(item))),
                 /* THEN */
            goto(of(P),T) : pi(tN, ?(now(tN)) :
               /* pi(wait, ?(wait $>=0) : pi(time, ?(time $= tN+wait) : */
            giveItems(P,tN) : goto(mo,tN) : leaveItems ),
                 /* ELSE */
            goto(mo,T) : pi(tN, ?(now(tN)) : deliverTo(P,tN))
```

```
        )
    )
).

proc(pickItems(P,T),
       if( some(n,some(t1,some(t2, mailPresent(P,n) & wantsCoffee(P,t1,t2) ))),
            /* THEN pickup both mail and cofee */
                   pickup(coffee,T) : pickup(mailTo(P),T),
            /* ELSE pickup only one item */
            if( some(n,mailPresent(P,n)),
                  /* THEN take only mail */
                pickup(mailTo(P),T),
                  /* ELSE take just coffee */
                pickup(coffee,T)
               )
          )
).

proc(giveItems(P,T),
       if( carrying(coffee) & carrying(mailTo(P)),
                /* Give both items */
            give(coffee,P,T) : give(mailTo(P),P,T),
                /* otherwise give just one of items */
              if( carrying(mailTo(P)),
                  give(mailTo(P),P,T),
                    give(coffee,P,T)
                )
         )
).

proc(leaveItems,
      while(some(item, carrying(item)),
             pi(item, ?(carrying(item)) :
                  pi(tN, ?(now(tN)) : putBack(item,tN) ))
             )
).

proc(deliverCoffee(P,T),
       if(robotLoc(mo),
       /* THEN */  if( carrying(coffee), serveCoffee(P,T),
                     pickup(coffee,T) : serveCoffee(P,T) ),
       /* ELSE */
         if(-carrying(coffee),
             goto(mo,T) :
               pi(t, ?(now(t)) : pickup(coffee,t) : serveCoffee(P,t)),
             serveCoffee(P,T)
           )
```

```
        )
    ).

/* Serve immediately on arrival */

proc(serveCoffee(P,T),
       if(robotLoc(of(P)),
           /* THEN */  give(coffee,P,T),
           /* ELSE */  goto(of(P),T) : pi(t, ?(now(t)) : give(coffee,P,t))
        )
    ).


/* Wait before serving */

proc(serveCW(P,T),
       if(robotLoc(of(P)),
           /* THEN */ pi(wait, ?(0 $<= wait) :
                          pi(t, ?(t $= T+wait) : give(coffee,P,t))),
           /* ELSE */  goto(of(P),T) :
                        pi(time, ?(0 $<= time) : pi(wait, ?(0 $<= wait) :
                             pi(t, ?(now(t)) : ?(time $= t+wait) :
                                  give(coffee,P,time) )))
        )
    ).


proc(deliverMail(P,T),
       if(robotLoc(mo),
       /* THEN */  if( carrying(mailTo(P)), serveMail(P,T),
                     pickup(mailTo(P),T) : serveMail(P,T) ),
       /* ELSE */
          if(-carrying(mailTo(P)),
             goto(mo,T) :
               pi(t, ?(now(t)) : pickup(mailTo(P),t) : serveMail(P,t)),
             serveMail(P,T)
            )
        )
    ).

proc(serveMail(P,T),
     if(robotLoc(of(P)),
         /* THEN */  give(mailTo(P),P,T),
         /* ELSE */  goto(of(P),T) : pi(t, ?(now(t)) : give(mailTo(P),P,t))
       )
   ).
```

```
/*
   This procedure is designed to test the mail delivery example from Section 5.6
   It does the same as the following procedure mail2 iff in the initial situatio
   only mailPresent(cr,1,s0) and mailPresent(fa,1,s0) are true
   (i.e., cr and fa do NOT want any coffee).
*/
proc(mail1,
 while( some(p, some(n, mailPresent(p,n) & -hasMail(p) & -status(p,out) )),
     pickBest(p, profs,
          ?( some(n, mailPresent(p,n)) & -status(p,out) & -hasMail(p) ) :
               pi(t, deliverMail(p,t) )   :
                  pi(t, ?(now(t)) : goto(mo,t)) : leaveItems
              )
      )
).

/* The procedure mail2 is supposed to find which sequence
           (mcr : mfa)    vs.    (mfa : mcr)
   of deliveries yields the highest total expected value.
   Note that the first literal in the termination condition of
   the while-loop in mail2 (or mail1) must be positive
   (to ground variables before negation-as-failure will be applied to them).
   The procedure mail2 delivers both mail and coffee (if requested);
   to test the mail delivery example from Section 5.6 make sure that
   in the initial situation only mailPresent(cr,1,s0) and mailPresent(fa,1,s0)
   are true (i.e., cr and fa do NOT want any coffee).
*/
proc(mail2,
 while( some(p, some(n, mailPresent(p,n) & -status(p,out) & -hasMail(p) )),
        pickBest(p, profs,
           ?(some(n, mailPresent(p,n) & -status(p,out) & -hasMail(p) )) :
                 pi(t, deliverTo(p,t) )
               )
       )
    ).

proc(mcr,       pi(t, deliverTo(cr,t) )
).

proc(mfa,    pi(t, deliverTo(fa,t) )
).

proc(goto(L,T),
  pi(rloc,?(robotLoc(rloc)) : pi(deltat,?(tT(rloc,L,deltat)) :
     goBetween(rloc,L,deltat,T)))).
```

```
/* Going from a location to the same location may take non-zero time if
   the location is a hallway. But normally, if a source and a destination are
   the same and the travel time between them is 0, then goBetween does nothing.
*/
proc(goBetween(Loc1,Loc2,Delta,T),
       if( Loc1=Loc2 & Delta=0, nil,
             startGo(Loc1,Loc2,T) :
             pi( time, ?(now(time)) :
                if( time=T,
                      % THEN (offline)
                    pi(t, ?(t $= T + Delta) : endGo(Loc1,Loc2,t) ) ,
                      % ELSE (online)
                    endGo(Loc1,Loc2,time)
                   )
                )
          )
).
/*
proc(goBetween(Loc1,Loc2,Delta,T),
       if( Loc1=Loc2 & Delta=0, nil,
             startGo(Loc1,Loc2,T) :
             pi(time, ?(time $= T + Delta) : endGo(Loc1,Loc2,time) )
          )
).
*/

             /*  Nondeterministic  Actions */

                    /* Robot's actions */

nondetActions(endGo(L1,L2,T),S,[endGoS(L1,L2,T),endGoF(L1,hall,TF)]) :-
         L1 \== hall,
           tT(L1, hall, Delta),
           start(S,TS), TF $= TS + Delta  ;
         /* OR */
         L1 == hall,
           start(S,TS), TF $= TS + 20.
/* A failed endGo action takes some time even if we started in the hall.
   This is not a very smart guess, though.*/

nondetActions(give(Item,Pers,T),S,[giveS(Item,Pers,T),giveF(Item,Pers,T)]).

/* Because a person loads things on robot, in this version the action
  'pickup' is deterministic.
*/

        /* Exogenous actions: the robot has no control over them */
```

```
nondetAction(cfRequest(P,T1,T2,T),S,
                    [cfRequest(P,T1,T2,T),noRequest(P,T)]).

nondetActions(mailArrives(Person,T),S,
                    [mailArrives(Person,T),noMail(Person,T)]).



/* Later : connect to the robot */

doSimul(A) :- agentAction(A), not senseAction(A), deterministic(A,S),
            printf("I'm doing the deterministic action %w\n",[A]).

doSimul(A) :- agentAction(A), nondetActions(A,S,NatOutcomesList),
            printf("I'm doing the stochastic action %w\n",[A]).

doSimul(A) :- senseAction(A), printf("Im doing the sensing action %w\n",[A]),
            printf("Type the value returned by the sensor\n",[]),
            stophere(yes),
            A =.. [ActionName,SensorName,Value,Time],
            read(Value).
/* Alternative implementation: introduce the predicate
value(SenseAction,Value) and for each sense action  senseAct
include in the domain axiomatization an axiom similar to
     value( senseAct(X,Val,Time), Val ).
Then we can simply call this predicate "value" in the clause above.
*/

        /* Identification conditions for robot's stochastic actions */

senseCondition(endGoS(Loc1,Loc2,T), robotLoc(Loc2) ).
senseCondition(endGoF(Loc1,Loc2,T), robotLoc(hall) ).
senseCondition(giveS(Item,Pers,T),W) :-
                       Item=mailTo(Pers), W = hasMail(Pers);
                       Item=coffee, W = hasCoffee(Pers).
senseCondition(giveF(Item,Pers,T),W) :-
                       Item=mailTo(Pers), W = (-hasMail(Pers));
                       Item=coffee, W = (-hasCoffee(Pers)).


/* Identification conditions for exogenous actions: not implemented */
/*
senseCondition(cfRequest(P,T1,T2,T),wantsCoffee(P,T1,T2)).
senseCondition(mailArrives(Person,T),
      some(now, some(prev, some(act,  curentSit(now) & now=do(act,prev) &
      some(mNow,                      mailPresent(Person,mNow) &
```

```
      some(mPrev,       mailPresent(Person,mPrev,prev) & mNow > mPrev )))))
).

%  What do we sense if nothing happened?
senseCondition(noRequest(P,T),
           -some(t1,some(t2,wantsCoffee(P,t1,t2))) ).

senseCondition(noMail(Person,T),
      some(now, some(prev, some(act,  curentSit(now) & now=do(act,prev) &
      some(mNow,                     mailPresent(Person,mNow) &
      some(mPrev,       mailPresent(Person,mPrev,prev) & mNow = mPrev )))))
).
*/
senseExo(S1,S2) :- S2=S1.

                          /* Probabilities */

                          /* Robot's actions */
prob(endGoS(Loc1,Loc2,T), PS, S) :- (PS is 0.99) .
prob(endGoF(Loc1,Loc2,T), PF, S) :- (PF is 0.01) .

prob(giveF(Item,Pers,T), Pr2, S) :-
             prob(giveS(Item,Pers,T), Pr1, S), Pr2 is (1 - Pr1).

/* If Ray/Craig  is in his office with probability _0.6_ and
  Visitor/Fahiem is in  with probability 0.6, then given
  our reward functions it is better to
  give(mailTo(fa),fa,45) and later give(mailTo(cr),cr,200)

 prob(giveS(Item,Pers,T), Pr, S) :-
                ( Pers=cr, (Pr is 0.6 ) ;
                   Pers=fa, (Pr is 0.6 ) ;
                   not Pers=cr, not Pers=fa, (Pr is 0.95)
                 ).
*/

/* If Ray/Craig is in his office with probability _0.8_ but
  Visitor/Fahiem is in his office with probability 0.6, then
 given our reward functions it is better
 to give(mailTo(cr),cr,110) and later give(mailTo(fa),fa,265)
*/
prob(giveS(Item,Pers,T), Pr, S) :-
                ( Pers=cr, (Pr is 0.8 ) ;
                  Pers=fa, (Pr is 0.6 ) ;
                  not Pers=cr, not Pers=fa, (Pr is 0.95) ).
```

```
                    /* Exogenous actions. */

/* The probabilistic model of how employees request coffee assumes that
   requests can occur in any situation (time of the request is tied up
   to the start time of the situation) and more often people ask
   for coffee if they did not have coffee yet.
   Variations: Prob depends on T that is not tied up to start time of S;
               some individuals more probably request coffee in certain
               time intervals (e.g., at the morning) than at other time,
               some employees request coffee more often than others, etc.

prob(cfRequest(Pers,T1,T2,T), Pro, S) :- person(Pers), start(S,T),
                frandom(Rand1), N is 100*Rand1,
                round(N, Bottom), T1 is 100 + T + Bottom,
                random(Rand2), M is 100*Rand2,
                round(M, Top), T2 is T1 + Top,
                   ( hasCoffee(Pers,S), Pro is (0.1) ;
                     not hasCoffee(Pers,S), Pro is (0.9) ).

prob(noRequest(Pers,T),P1,S) :- prob(cfRequest(Pers,T1,T2,T), P2, S),
                P1 is 1 - P2.

prob(mailArrives(Pers,T), Pro, S) :- person(Pers),
                    start(S,TS),        % Assume that TS is a fixed time %
                    Lambda is (T - TS)/10, exp(Lambda,Res),
                    Pro is 1 - (1/Res).
prob(noMail(Pers,T),P1,S) :-
                     prob(mailArrives(Pers,T), P2, S), P1 is 1 - P2.
*/

               /* Preconditions for Primitive Actions */

                       /* Robot's actions */

poss(pickup(Item,T),S) :- Item = coffee,
         not carrying(Item,S),  robotLoc(mo,S).

poss(pickup(Item,T),S) :- Item = mailTo(P),
         not carrying(Item,S),  robotLoc(mo,S), mailPresent(P,N,S), N > 0.

poss(giveS(Item,Person,T),S) :- Item=coffee,
       carrying(Item,S), robotLoc(of(Person),S).

/* The robot is permitted to give mail only to the addressee */
poss(giveS(Item,Person,T),S) :- Item=mailTo(Person),
       carrying(Item,S), robotLoc(of(Person),S).
```

```
poss(giveF(Item,Person,T),S) :- Item=coffee,
       carrying(Item,S), robotLoc(of(Person),S).

poss(giveF(Item,Person,T),S) :- Item=mailTo(Person),
       carrying(Item,S), robotLoc(of(Person),S).

poss(startGo(Loc1,Loc2,T),S) :- robotLoc(Loc1,S), not going(L,LL,S).

poss(endGoS(Loc1,Loc2,T),S) :- going(Loc1,Loc2,S).
poss(endGoF(Loc1,Loc2,T),S) :- going(Loc1,Loc3,S), not (Loc2=Loc3).

poss(putBack(Item,T),S) :-  robotLoc(mo,S), carrying(Item,S).

                   /*  Exogenous actions  */
/*
poss(mailArrives(Person,T),S).
poss(noMail(P,T),S).

poss(cfRequest(Person,T1,T2,T),S) :-  T $<= T1, T1 $< T2.
poss(noRequest(P,T),S).
*/

               /*   Successor State Axioms     */


hasCoffee(Person,do(A,S)) :- A = giveS(coffee,Person,T).

hasCoffee(Person,do(A,S)) :- A = sense(buttons,Answer,T), Answer==1,
                             robotLoc(of(Person),S).

hasCoffee(Person,do(A,S)) :-  hasCoffee(Person,S),
                              not A = giveS(coffee,Person,T),
                              not A = sense(buttons,1,T).

hasMail(Person, do(A,S)) :-  A = giveS(mailTo(Person),Person,T).

hasMail(Person, do(A,S)) :-  A = sense(buttons,Answer,T), Answer==1,
                             robotLoc(of(Person),S).

hasMail(Person, do(A,S)) :-  hasMail(Person,S),
                             not A = sense(buttons,1,T),
                             not A = giveS(mailTo(Person),Person,T).

robotLoc(L,do(A,S)) :-   A = endGoS(LocStart,L,T).

robotLoc(L,do(A,S)) :-   A = endGoF(LocStart,L,T).
```

```
robotLoc(L,do(A,S)) :-   A = sense(coordinates,V,Time),
                       xCoord(V,X), yCoord(V,Y), inside(L,X,Y).

robotLoc(L,do(A,S)) :-   robotLoc(L,S),
              not A = endGoS(Loc2,Loc3,T),
              not A = endGoF(Loc2,Loc3,T),
              not ( A = sense(coordinates,V,T),
                    xCoord(V,X), yCoord(V,Y), inside(Loc,X,Y), L \== Loc ).

inside(mo,X,Y) :-
           bottomY(mo,Yb),  Yb =< Y,
           topY(mo,Yt),  Y =< Yt,
           leftX(mo,Xl), Xl =< X,
           rightX(mo,Xr),  X =< Xr.

inside(of(Person),X,Y) :-
           bottomY(of(Person),Yb), Yb =< Y,
           topY(of(Person),Yt), Y =< Yt,
           leftX(of(Person),Xl), Xl =< X,
           rightX(of(Person),Xr), X =< Xr.

inside(hall,X,Y) :- not inside(mo,X,Y), not inside(of(P),X,Y).


going(Origin,Destination,do(A,S)) :- A = startGo(Origin,Destination,T).

going(Origin,Destination,do(A,S)) :- going(Origin,Destination,S),
                   not A = endGoS(Origin,Destination,T),
                   not A = endGoF(Origin,Elsewhere,T).


carrying(Item,do(A,S)) :- A = pickup(Item,T).

carrying(Item,do(A,S)) :-  carrying(Item,S),
                           not A = giveS(Item,Person,T),
                           not A = putBack(Item,T).

/*  status(Person,DoorPersonOfficeStatus, Sit) means whether the door of
Person's office is open (and then Person is in), closed (Person is out)
or unknown (if the office of Person has never been visited before).
*/
status(Person,Stat,do(A,S)) :-
              A = giveS(Item,Person,T), Stat=in. % Person is in

status(Person,Stat,do(A,S)) :-
              A = giveF(Item,Person,T), Stat=out. % Person is out
```

```
status(Person,Stat,do(A,S)) :-  status(Person,Stat,S),
                        not A=giveS(Item,Person,T),
                        not A=giveF(Item,Person,T).


/* wantsCoffee(P,T1,T2,do(A,S)) :-  A = cfRequest(P,T1,T2,T). */

wantsCoffee(P,T1,T2,do(A,S)) :-  wantsCoffee(P,T1,T2,S),
                            not A=giveS(coffee,P,T).


/* The following axiom assumes that if initially a person P does not
   have mail in the mailbox, this is represented by  leaving out
   mailPresent(P,X,s0); it is also assumed that mail arrives one by one.

mailPresent(Person,N,do(A,S)) :-
           A=mailArrives(Person,T), mailPresent(Person,M,S), M > 0, N is M+1.
mailPresent(Person,N,do(A,S)) :-
           A=mailArrives(Person,T), not mailPresent(Person,M,S), N is 1.
*/

mailPresent(Person,N,do(A,S)) :-  A = putBack(mailTo(Person),T), N is 1 .
                  %         not mailPresent(Person,X,S), N is 1.

% mailPresent(Person,N,do(A,S)) :-  A = putBack(mailTo(Person),T),
%                           mailPresent(Person,M,S), N is M + 1.

mailPresent(Person,N,do(A,S)) :-  mailPresent(Person,N,S),
                         not A = pickup(mailTo(Person),T).


                         /* Reward function */

reward(R, s0) :- R $= 0.
/* A simpler version might also be sufficient in most cases:
    reward(0, s0).
*/

reward(V, do(A,S)) :- A = giveS(coffee,Person,T),
             wantsCoffee(Person,T1,T2,S),
               V $<= (T2 - T)/2,
              V $<= T - (3*T1 - T2)/2,
               VL $<= V, rmax(VL).

reward(V, do(A,S)) :- A = giveS(coffee,Person,T),
             not wantsCoffee(Person,T1,T2,S),
             T $<= TM, rmin(TM),
```

```
            V $= 0.

/* Just to give an idea what rewards the robot may get for serving coffee:

wantsCoffee(ray,340,380).     V $<= (380 - T)/2 & V $<= T - 320. rmax(V)=20
wantsCoffee(vi,150,200). V $<= (200 - T)/2 & V $<= T - 125. rmax(V)=25
*/

/* We use a declining linear function of time as a reward function
   to encourage earlier mail delivery. */

reward(V, do(A,S)) :- A = giveS(mailTo(P),P,T1),
    %  whenMailArrived(P,S,N,T2),
     ( P=cr,   V $<= 30 - T1/10 ;
       P=fa,   V $<= 15 - T1/20 ;
       P\==cr, P\==fa,  V $<= 10 - T1/10
      ),
     rmax(V).


/* When the robot gets stuck in the hall, it costs 5 points ;
   other actions bring 0 costs/rewards.
reward(V, do(A,S)) :-
        A = startGo(Loc1,Loc2,T), V $= 0 ;
          A = endGoS(Loc1,Loc2,T),
          tT(Loc1,Loc2,T12), V $= 0 ;  % (1000 - T)/ T12 % ;
        A = endGoF(Loc1,Loc2,T),
          V $= -5.

reward(V, do(A,S)) :-   not (A=giveS(Item,P,T), A=startGo(L1,L2,T),
                                 A=endGoS(L1,L2,T), A=endGoF(L1,L2,T) ),
                        V $= 0.
*/

reward(V, do(A,S)) :-   not  A=giveS(Item,P,T),  V $= 0.



whenMailArrived(P,s0,N,Time)  :-  mailPresent(P,N,s0), N > 0, Time $= 0.
whenMailArrived(P,S2,N,Time) :- S2=do(A,S1),
       ( A=mailArrives(P,T), Time $= T,
                ( mailPresent(P,M,S1) -> N is M+1 ; N=1) ;
         not A=mailArrives(P,T),  whenMailArrived(P,S1,N,Time)
        ).

/* The time of an action occurrence is its last argument. */
```

```
time(pickup(Item,T),T).
time(giveS(Item,Person,T),T).
time(giveF(Item,Person,T),T).
time(startGo(Loc1,Loc2,T),T).
time(endGoS(Loc1,Loc2,T),T).
time(endGoF(Loc1,Loc2,T),T).
time(putBack(Item,T),T).
time(sense(buttons,Answer,Time),Time).
time(sense(coordinates,V,Time),Time).
% time(cfCancel(P,T1,T2,T),T).
% time(cfRequest(Pers,T1,T2,T),T).
% time(noRequest(P,T), T).
% time(mailArrives(P,T),T).
% time(noMail(P,T), T).

/* Restore situation arguments to fluents. */

restoreSitArg(robotLoc(Rloc),S,robotLoc(Rloc,S)).
restoreSitArg(hasCoffee(Person),S,hasCoffee(Person,S)).
restoreSitArg(hasMail(Person),S,hasMail(Person,S)).
restoreSitArg(going(Loc1,Loc2),S,going(Loc1,Loc2,S)).
restoreSitArg(carrying(Item),S,carrying(Item,S)).
restoreSitArg(status(Pers,Office),S,status(Pers,Office,S)).
restoreSitArg(wantsCoffee(Person,T1,T2),S,wantsCoffee(Person,T1,T2,S)).
restoreSitArg(mailPresent(Person,N),S,mailPresent(Person,N,S)).
restoreSitArg(currentSit(X),S,currentSit(X,S)).
/* This is a handy expression */
currentSit(X,S) :- X = S.


                        /* Primitive Action Declarations */

agentAction(pickup(Item,T)).
agentAction(give(Item,Person,T)).
agentAction(startGo(Loc1,Loc2,T)).
agentAction(endGo(Loc1,Loc2,T)).
agentAction(putBack(Item,T)).
%   agentAction(cfRequest(Pers,T1,T2,T)).
%   agentAction(mailArrives(Pers,T)).
%   agentAction(cfCancel(P,T1,T2,T)).


/* All sensing actions have 3 arguments:
   - the 1st argument is property/quantity that we would like to measure
   - the 2nd argument is the value returned by sensor at the run-time
   - the last 3rd argument is time
*/
```

```
senseAction(sense(buttons,Answer,Time)).  /* Answer can be 1 or 0 */
senseAction(sense(coordinates,V,Time)).

differentiatingSeq(endGo(Loc1,Loc2,T), sense(coordinates,V,T)).
differentiatingSeq(give(Item,Person,T), sense(buttons,Answer,T)).


                             /* Initial Situation. */

% robotLoc(park,s0).
robotLoc(mo,s0).
status(ray,unknown,s0).
status(itrc,unknown,s0).
%   office lp269
status(lg,unknown,s0).
%   office lp271
status(ma,unknown,s0).
status(st,unknown,s0).
status(yv,unknown,s0).
%  office lp276
status(fa,unknown,s0).
status(vi,unknown,s0).
%  office lp290b
status(cr,unknown,s0).


% wantsCoffee(ray,340,380,s0).
/*
wantsCoffee(yv,800,850,s0).
wantsCoffee(fa,500,550,s0).
wantsCoffee(lg,150,200,s0).
wantsCoffee(vi,1100,1150,s0).
wantsCoffee(cr,310,400,s0).
wantsCoffee(st,600,650,s0).
wantsCoffee(ma,1000,1050,s0).
*/

/* times for video: */
/*
wantsCoffee(fa,150,200,s0).
wantsCoffee(yv,320,400,s0).
wantsCoffee(ma,320,400,s0).
wantsCoffee(lg,230,300,s0).
wantsCoffee(cr,310,400,s0).
*/
% mailPresent(lg, 1,s0).
% mailPresent(ray,1,s0).
```

```
% mailPresent(vi,1,s0).
mailPresent(cr,1,s0).
mailPresent(fa,1,s0).

/* A simple example from Section 5.3: run the procedure deliver */
% wantsCoffee(ann,100,150,s0).
% mailPresent(joe,4,s0).
%
% tT0( mo,    of(ann),    45 ).
% tT0( mo,    of(joe),    100 ).
% tT0( mo,    of(bill),   50 ).

tT(L,L,0) :- L \== hall.
tT(L1,L2,T) :- tT0(L1,L2,T) ;
                       tT0(L2,L1,T).

/*
--- TABLE OF AVERAGE TRAVEL TIME FROM cm ---

        -----------RAY    75  (51, 72 cm->ray; 56, 55 ray->cm)
        |
        +--------------
        |
        |
        +--ITRC            24
    CM--+
        |
        |
        |
        +--VISITOR         40
        |
 LOUNGE-+                 58 (cm->lg), 63 (lg->cm)
        |
  ------+--------------------
*/

tT0( park,   mo,       100 ).     /* 73, 82 on simulator */
tT0( park,   hall,      80 ).
tT0( hall,   mo,       20 ).
tT0( hall,   hall,      10 ).

tT0( mo,    of(ray),    110 ).   /* 51,72,56,55 on simulator*/
tT0( mo,    of(cr),    110 ).   /* 51,72,56,55 on simulator*/
tT0( mo,    of(itrc),    30 ).   /* 24 on simulator */
tT0( mo,    of(vi),  45 ).   /* 29 on simulator  */
tT0( mo,    of(fa),  45 ).   /* 29 on simulator  */
tT0( mo,    of(lg),  75 ).   /* 58, 63 on simulator */
```

```
tT0( mo,    lp271,  70 ).
tT0( mo,    lp276,  45 ).
tT0( mo,    of(st),  70 ).
tT0( mo,    of(yv),  70 ).
tT0( mo,    of(ma),  70 ).

tT0( hall,    of(ray),  40 ).
tT0( hall,    of(cr),    60 ).   /* 51,72,56,55 on simulator*/
tT0( hall,    of(itrc),    20 ).   /* 24 on simulator */
tT0( hall,    of(vi),  20 ).   /* 29 on simulator  */
tT0( hall,    of(fa),  20 ).   /* 45 on simulator  */
tT0( hall,    of(lg),  35 ).   /* 58, 63 on simulator */
tT0( hall,    of(st),  30 ).
tT0( hall,    of(yv),  30 ).
tT0( hall,    of(ma),  30 ).
tT0( hall,    lp271,  30 ).
tT0( hall,    lp276,  20 ).

/* Other travel times (measured on simulator):  */

tT0( of(cr),    of(itrc),    70 ).
tT0( of(cr),    of(vi),  120 ).
tT0( of(cr),    of(fa),  120 ).
tT0( of(cr),    of(lg),  130 ).
tT0( of(lg),  of(itrc),    70 ).
tT0( of(lg),  of(vi),  60 ).
tT0( of(lg),  of(fa),   60 ).
tT0( of(vi),  of(itrc),    50 ).
tT0( of(fa),   of(itrc),  120 ).
tT0( park,        of(vi),  60 ).


/* Geometric coordinates on the real of map. */
/* New coordinates shifted by -800 in the first coordinate:
  of(ray) = (3753.4, 1800)     cm = (2675, 2555)      park = (118, 2487)
       OLD= (4530, 1750)      OLD= (3465, 2600)      OLD= (930, 2530)
*/

xCoord([X,Y],R) :- R=X.
yCoord([X,Y],R) :- R=Y.

xCoord(V,X) :- V=[X,Y,Angle].
yCoord(V,Y) :- V=[X,Y,Angle].

/*                       X    Y                              X    Y
  left_bottom(mo)  =  (2400,2550)     right_top(mo)  =  (2780,2900)
  left_bottom(fa)  =  (1600,2300)     right_top(fa)  =  (1800,2520).
```

```
  left_bottom(cr)  =  (3700,1600)     right_top(cr)  =  (3850,1800)
  left_bottom(lg)  =  (550,2600)      right_top(lg)  =  (800,2800)
  left_bottom(itrc) =  (3100,2300)    right_top(itrc) = (3250,2500)
  left_bottom(lp271) = (900,2550)     right_top(lp271)= (1200,2800)
  left_bottom(elevator)=(3250,1600)   right_top(elevator) = (3440,1650)
*/

bottomY(mo, 2520).
% bottomY(lp271, 2600).  % bottomY(lp276, 2300).
bottomY(of(st), 2550).
bottomY(of(yv), 2550).
bottomY(of(ma), 2550).
bottomY(of(cr), 1600).
bottomY(of(fa), 2300).
bottomY(of(vi), 2300).
bottomY(of(lg), 2600).
bottomY(of(itrc), 2300).

topY(mo, 2900).
% topY(lp271, 2800).    % topY(lp276, 2500).
topY(of(st), 2800).
topY(of(yv), 2800).
topY(of(ma), 2800).
topY(of(cr), 1800).
topY(of(fa), 2520).
topY(of(vi), 2520).
topY(of(lg), 2800).
topY(of(itrc), 2500).

leftX(mo, 2400).
% leftX(lp271, 900).    % leftX(lp276, 1600).
leftX(of(st), 900).
leftX(of(yv), 900).
leftX(of(ma), 900).
leftX(of(cr), 3700).
leftX(of(fa), 1600).
leftX(of(vi), 1600).
leftX(of(lg), 550).
leftX(of(itrc), 3100).

rightX(mo, 2780).
%  rightX(lp271, 1200).  %  rightX(lp276, 1800).
rightX(of(st), 1200).
rightX(of(yv), 1200).
rightX(of(ma), 1200).
rightX(of(cr), 3850).
rightX(of(fa), 1800).
```

```
rightX(of(vi), 1800).
rightX(of(lg), 800).
rightX(of(itrc), 3250).


drivePath( _,    of(lg),  [ ( 840, 2560 ), ( 770, 2530 ) ] ).
drivePath( _,    of(vi), [ ( 1670, 2450 ) ] ).
drivePath( _,    of(fa), [ ( 1670, 2450 ) ] ).
drivePath( of(itrc), mo,      [ ( 3120, 2450 ), ( 2700, 2550 ) ] ).
drivePath( mo,   of(itrc),    [ ( 2665, 2620 ), ( 3120, 2450 ) ] ).
drivePath( of(cr),  mo,
      [ ( 3760, 2420 ), (3560, 2480 ), ( 2820, 2560 ), ( 2690, 2550 ) ] ).
drivePath( _,    of(st),  [(1500,2500), (1030,2600), (1010,2625)] ).
drivePath( _,    of(ma),  [(1500,2500), (1030,2600), (1010,2625)] ).
drivePath( _,    of(yv),  [(1500,2500), (1030,2600), (1010,2625)] ).


drivePath( _,    mo,      [ (2685, 2500), (2700, 2630) ] ).
drivePath( _,    of(itrc),   [ ( 3120, 2450 ) ] ).
drivePath( _,    of(cr),   [ (3535, 2490 ), (3760, 2480), ( 3760, 1720 ) ] ).
drivePath(_,  park, [(200, 2500)] ).

driveSim( StartPos, EndPos ) :- nl, write("drive, drive, drive from "),
                              write(StartPos), write(" to "), write(EndPos), nl.

/* Drive in the corridor and turn to the goal location */

driveReal( StartPos, EndPos ) :-
drivePath( StartPos, EndPos, Path ),   % get path
           hli_go_path( Path ),               % drive, drive...
look( EndPos, X, Y ),              % get aim point for turning
hli_turn_to_point( X, Y ).         % and turn the robot


look( of(lg),  640, 2670 ).
look( of(vi), 1600, 2180 ).
look( of(fa), 1600, 2180 ).
look( mo,      2700, 2800 ).
look( of(itrc),    2872, 2200 ).
look( of(cr),     3728, 1650 ).
look( of(st),  1030, 2750 ).
look( of(ma),  1030, 2750 ).
look( of(yv),  1030, 2750 ).
```

Appendix D

An Online Decision-Theoretic Golog

An interpreter enclosed in this section works with actions that have temporal arguments. Because it includes off-line interpreter, this off-line interpreter can be used to run all examples that include actions with time.

D.1 An On-Line Interpreter

```
/*******************************************************************
      An On-line Decision Theoretic Golog Interpreter based on
        An Incremental Decision Theoretic Golog Interpreter
                          May, 2000.

            Do not distribute without permission.
            Include this notice in any copy made.

Permission to use, copy, and modify this software and its documentation
for non-commercial research purpose is hereby granted without fee,
provided that this permission notice appears in all copies. This
software cannot be used for commercial purposes without written permission.
This software is provided "as is" without express or implied warranty
(including all implied warranties of merchantability and fitness).
No liability is implied for any damages resulting from
or in connection with the use or performance of this software.

E-mail questions/comments about the interpreter to Mikhail Soutchanski:
            mes [at] cs [dot] toronto [dot] edu


NOTICE: this software works only with Eclipse Prolog 3.5.2 available from
        IC-PARK (Imperial College, London, UK) because this interpreter
        relies on a linear programming solver built in Eclipse Prolog 3.5.2.
        The solver finds solutions for a system of linear inequalities
        and for linear programming problems only if all coefficents
        are *rational* numbers. Contact http://www.icparc.ic.ac.uk/eclipse/
```

```
        for a licence (free to academic and research institutions).
      More recent versions of Eclipse Prolog use different set of predicates.

*********************************************************************/


:- lib(r).
:- dynamic(proc/2).            /* Compiler directives. Be sure   */
:- set_flag(all_dynamic, on).  /* that you load this file first! */
:- set_flag(print_depth,500).
/* a toolbox of debugging tricks: put stophere(yes) where necessary. */
:- pragma(debug).
:- dynamic(stophere/1).
:- set_flag(stophere/1, spy, on).      %  type debug and hit "l" to leap
:- set_flag(stophere/1, leash, stop).  %  from one break-point to another


:- op(800, xfy, [&]).   /* Conjunction */
:- op(850, xfy, [v]).   /* Disjunction */
:- op(870, xfy, [=>]).  /* Implication */
:- op(880,xfy, [<=>]).  /* Equivalence */
:- op(950, xfy, [:]).   /* Action sequence */
:- op(970, xfy, [#]).   /* Nondeterministic action choice */

/* The predicate best() is the top level call.
  Add an end-of-program marker "nil" to the tail of program expression E ,
   compute the best policy starting from s0. Call this predicate to start
  online execution of a Golog program.
*/

best(E,H,Pol,V,File) :- get_flag(unix_time, StartT),
          online(E : nil,s0,H,Pol,V),
          Val is float(V),
          open( File, append, Stream),
          date(Date),
          printf(Stream, "\n\n    This report is started at time %w\n", [Date]),
         ( proc(E,Body) ->
           printf(Stream, "The Golog program is\n proc(%w,\n  %w)\n",[E,Body]) ;
           printf(Stream, "The Golog program is\n %w\n",[E])
          ),
          get_flag(unix_time, EndT), Elapsed is EndT - StartT,
          printf("\nThe computation took %w seconds of unix time",[Elapsed]),
          printf(Stream, "\nTime elapsed is %w seconds\n", [Elapsed]),
  %          printf(Stream, "The optimal policy is \n %w \n", [Pol]),
          printf(Stream, "The value of the optimal policy is %w\n",[Val]),
          close(Stream).
```

```
/*
This predicate is useful only for testing the offline incrBestDo interpreter.
*/
bp(E,H,Pol,Val,Prob,File) :-
          cputime(StartT), %%%get_flag(unix_time, StartT),
          incrBestDo(E : nil,s0,H,ER,Pol,V,Prob),
          Val is float(V),
          open( File, append, Stream),
          date(Date),
          printf(Stream, "\n\n    This report is started at time %w\n", [Date]),
         ( proc(E,Body) ->
           printf(Stream, "The Golog program is\n proc(%w,\n  %w)\n",[E,Body]) ;
           printf(Stream, "The Golog program is\n %w\n",[E])
          ),
          cputime(EndT),  %%%get_flag(unix_time, EndT),
          Elapsed is 1.0*(EndT - StartT),
    printf("\nThe computation took %w seconds of unix time",[Elapsed]),
        printf(Stream, "\nTime elapsed is %8.4g seconds\n", [Elapsed]),
          printf(Stream, "The optimal policy is \n %w \n", [Pol]),
          printf(Stream, "The value of the optimal policy is %w\n",[Val]),
printf(Stream,"The probability of successful termination of this policy is %w\n"
          close(Stream).



/*  online(Expr,S,H,Pol,V) means the following.
  Given a Golog program Expr, a situation S and horizon H find an
  optimal sequence of actions Pol with an accumulated expected value V.
  In logical terms, the problem is

   DomainAxioms \=?=  (\exists p,v) online(Expr : Nil, S0,H,p,v)

  and any binding for existentially quantified variables "p","v"
  obtained as a side-effect of solving this entailment task constitues
  a solution of a decision theoretical problem.

  incrBestDo(E,S,H,ER,Pol,V,Prob) means the following.
  Given a Golog program expression E, situation S and horizon H find
  a policy Pol of the highest expected value V and find the sub-program ER
  of E that remains after executing the first action from the optimal
  policy Pol. The probability Prob is the total probability of all those
  alternative branches in Pol which have leaves different from Stop, where
  Stop is a zero-cost action marking a prematurely terminated program E.
  The program ER is the result of a one-step transition from E along the
  optimal policy Pol. The horizon H must be a non-negative integer number.
*/
```

```
online(E,S,H,Pol,U) :-   incrBestDo(E,S,H,ER,Pol1,U1,Prob1),
     %  stophere(yes),
     ( final(ER,S,H,Pol1,U1), Pol=Pol1, U=U1 ;
       reward(V,S),
       ( ER \== nil, Pol1 = (A : Rest) ;  ER == nil, Pol1 = A ),
       (
         (agentAction(A), deterministic(A,S),
            doReally(A),     /* do A in reality or doSimul(A) - ask user*/
            senseExo(do(A,S),Sg),
            decrement(H , Hor),
            !,
            online(ER,Sg,Hor,PolRem,URem),
            Pol=(A : PolRem), U $= V + URem
          ) ;
         (
          (senseAction(A),
             doReally(A),     /* do sensing in reality or in simulation */
             senseExo(do(A,S),Sg),
             decrement(H , Hor),
             !,
             online(ER,Sg,Hor,PolRem,URem),
             Pol=(A : PolRem), U $= V + URem
           ) ;
          (agentAction(A), nondetActions(A,S,NatOutcomes),
             doReally(A),     /* do A in reality or doSimul(A) - ask user*/
             senseEffect(A,S,NatOutcomes,SEff),
                              /* SEff results from real sensing */
             senseExo(SEff,Sg),
             decrement(H , Hor),
             !,
             online(ER,Sg,Hor,PolRem,URem),
             Pol=(A : PolRem), U $= V + URem
           )
          )
        )
      ).

doReally(A) :- doSimul(A).

/* Later : connect to the robot */

doSimul(A) :- agentAction(A), not senseAction(A), deterministic(A,S),
            printf("I'm doing the deterministic action %w\n",[A]).

doSimul(A) :- agentAction(A), nondetActions(A,S,NatOutcomesList),
            printf("I'm doing the stochastic action %w\n",[A]).
```

```
doSimul(A) :- senseAction(A), printf("Im doing the sensing action %w\n",[A]),
            printf("Type the value returned by the sensor\n",[]),
       %    stophere(yes),
            A =.. [ActionName,SensorName,Value,Time],
            read(Value).
/* Alternative implementation: introduce the predicate
value(SenseAction,Value) and for each sense action  senseAct
include in the domain axiomatization an axiom similar to
     value( senseAct(X,Val,Time), Val ).
Then we can simply call this predicate "value" in the clause above.
*/


/* In the simplest case, there are no exogenous actions. */

senseExo(S1,S2) :- S2=S1.


/* Domain model includes a pre-determined sequence Seq of sensing actions
   that have to be executed after stochastic action A. The syntax is
   the following (in the case when 3 sensing actions must be executed):
      differentiatingSeq(A, SenseAction1 : (SenseAction2 : SenseAction3)).
   Each sensing action has 3 arguments: the 1st argument is the constant
   representing what has to be sensed, the 2nd argument represents
   a term (variable) that gets bounded to a certain value returned
   from a sensor at the run-time, the 3rd argument represents the moment
   of time when the sensing action is performed.
*/

senseEffect(A,S,NatOutcomes,SE) :-
          differentiatingSeq(A,Seq),
          getSensorInput(do(X,S),Seq, SE),
             /* find X: an unknown outcome of the stochastic action A */
          diagnose(S,SE,NatOutcomes,X).

getSensorInput(S,A, SE) :-  senseAction(A),
       doReally(A),  /* connect to sensors and get data */
       % doSimul(A)  /* ask user for a sensor value when A is simulated*/
       SE=do(A,S).

getSensorInput(S,Seq, SE) :- Seq = (A : Tail), senseAction(A),
       doReally(A),   /* connect to sensors and get data */
       % doSimul(A)   /* ask user */
       getSensorInput(do(A,S),Tail, SE).
```

```
/*
  diagnose(S,SE,OutcomesList,N) is true iff N is Natura's action that
  actually happened in S and this action belongs to the list of outcomes
  which might occur in S. It is assumed that the situation SE contains
  all sensing information required to disambiguate between different
  possible outcomes.
  Domain model includes a set of statements that characterize mutually
  exclusive conditions that need to be evaluated to determine which
  outcome occurred in reality. These statements have the following syntax:
      senseCondition(N, phi).
  where phi is an arbitrary pseudo-fluent expression.
*/

diagnose(S,SEff,[N],X) :- senseCondition(N,C),
             ( X=N, holds(C,SEff)  /* nature did the action N */;
               write("Failed to identify an outcome")
              ).

diagnose(S,SEff,[N, NN | Outcomes],X) :-
  senseCondition(N,C), /* assume that N happened */
  ( X=N, holds(C,SEff) /* verified that nature did action N */ ;
    diagnose(S,SEff,[NN | Outcomes],X) /*if not, consider other outcomes*/
   ).
```

```
incrBestDo((E1 : E2) : E,S,H,ER,Pol,V,Prob) :-   positive(H),
           incrBestDo(E1 : (E2 : E),S,H,ER,Pol,V,Prob).

incrBestDo(?(C) : E,S,H,ER,Pol,V,Prob) :-    positive(H),
          holds(C,S) -> incrBestDo(E,S,H,ER,Pol,V,Prob) ;
          (ER=nil, (Prob is 0.0) , Pol = stop, reward(V,S)).

incrBestDo(pi(X,E1) : E,S,H,ER,Pol,V,Prob) :-   positive(H),
            sub(X,_,E1,E1_X),
            incrBestDo(E1_X : E,S,H,ER,Pol,V,Prob).

incrBestDo((E1 # E2) : E,S,H,ER,Pol,V,Prob) :-    positive(H),
       incrBestDo(E1 : E,S,H,ER1,Pol1,V1,Prob1),
       incrBestDo(E2 : E,S,H,ER2,Pol2,V2,Prob2),
       ( lesseq(V1,Prob1,V2,Prob2) ->
                 (ER=ER2, Pol=Pol2, Prob=Prob2, V $= V2) ;
                 (ER=ER1, Pol=Pol1, Prob=Prob1, V $= V1)
        ).
```

```
incrBestDo(if(C,E1,E2) : E,S,H,ER,Pol,V,Prob) :-   positive(H),
             holds(C,S) -> incrBestDo(E1 : E,S,H,ER,Pol,V,Prob) ;
                           incrBestDo(E2 : E,S,H,ER,Pol,V,Prob).

incrBestDo(while(C,E1) : E,S,H,ER,Pol,V,Prob) :-   positive(H),
            holds(-C,S) -> incrBestDo(E,S,H,ER,Pol,V,Prob) ;
                 incrBestDo(E1 : while(C,E1) : E,S,H,ER,Pol,V,Prob).

incrBestDo(ProcName : E,S,H,ER,Pol,V,Prob) :-   positive(H),
              proc(ProcName,Body),
              incrBestDo(Body : E,S,H,ER,Pol,V,Prob).

/*
  Discrete version of pi. pickBest(x,f,e) means: choose the best value
   of x from the finite range of values f, and for this x,
   do the complex action e.
*/
incrBestDo(pickBest(X,F,E) : Tail,S,H,ER,Pol,V,Prob) :-  positive(H),
     range(F,R),
     ( R=[D],     sub(X,D,E,E_D),
                    incrBestDo(E_D : Tail,S,H,ER,Pol,V,Prob)           ;
       R=[D1,D2], sub(X,D1,E,E_D1), sub(X,D2,E,E_D2),
                    incrBestDo((E_D1 # E_D2) : Tail,S,H,ER,Pol,V,Prob)  ;
       R=[D1,D2 | List],  List=[D3 | Rest],
                  sub(X,D1,E,E_D1), sub(X,D2,E,E_D2),
     incrBestDo((E_D1 # E_D2 # pickBest(X,List,E)) : Tail,S,H,ER,Pol,V,Prob)
      ).


incrBestDo(A : E,S,H,ER,Pol,V,Prob) :-   positive(H),
      agentAction(A), deterministic(A,S),
      ( not poss(A,S), ER=nil, Pol=stop, (Prob is 0.0) ,
                     reward(V,S);
        poss(A,S),
        decrement(H , Hor),
        start(S,T1),  time(A,T2),  T1 $<= T2,
        ER = E,
        incrBestDo(E,do(A,S),Hor,_,RestPol,VF,Prob),
        reward(R,S),
        V $= R + VF,   %%% if discount is 0.9: V $= R + (9/10)*VF
        ( RestPol == nil, Pol = A ;
          RestPol \== nil,  Pol = (A : RestPol)
         )
       ).


incrBestDo(A : E,S,H,ER,Pol,V,Prob) :-   positive(H),
```

```
        agentAction(A), nondetActions(A,S,NatOutcomesList),
        decrement(H , Hor),
        bestDoAux(NatOutcomesList,E,S,Hor,RestPol,VF,Prob),
        /* RestPol can be stop or a conditional policy */
        reward(R,S),
    ( RestPol = ( ?(_) : stop ),  ER=nil,  Pol=(A : stop),  V $= R ;
     RestPol \= ( ?(_) : stop ),  ER=E,    Pol=(A : RestPol),
           V $= R + VF   %%% if discount is 0.9: V $= R + (9/10)*VF
         ).

/* Because the decision tree built by this interpreter grows quickly
   with increase in horizon, it is computationally advantageous
   to compute optimal policies from sub-parts of a large Golog program
   and then piece computed sub-policies together into a policy
   for a whole program. To accomplish this, the programmer can
   use constructs which allow to indicate which sub-parts of
   the program can be used individually for computing sub-policies.
   The following construct loc(Prog1) : Prog2 means that a policy Pol1
   can be computed offline given Prog1 and used later to compute offline
   an optimal policy Pol from the program (Pol1 : Prog2). Note that
   Pol1 is a conditional Golog program without nondeterministic choices.
   Once the policy Pol1 is computed, the construct loc() will not be
   applied later to compute policies from the program that remains
   after executing on-line the first action in the policy Pol.
   Note that Pol1 may have several occurrences of "nil" and "stop".
*/
 incrBestDo(loc(E1) : E,S,H,ER,Pol,V,Prob) :-   positive(H),
             incrBestDo(E1 : nil,S,H,_,Pol1,V1,Prob1),
             incrBestDo(Pol1 : E,S,H,ER,Pol,V,Prob).

/* In the next version the locality operation persists
   until the locally optimized program does not finish on-line.
   The second difference from the previous operator is that
   a policy for on-line execution is computed without taking
   into account the program expression E that follows after
   a locally optimized expression E1.
*/

/* Insert once the end-of-program marker "nil" after
   the program expression E1 and use "optimize" instead of "limit"
   to indicate that "nil" has beed already added after E1.
*/

incrBestDo(limit(E1) : E,S,H,ER,Pol,V,Prob) :-   positive(H),
        incrBestDo(optimize(E1 : nil) : E,S,H,ER,Pol,V,Prob).
```

```
incrBestDo(optimize(Expr1) : E,S,H,ER,Pol,V,Prob) :-   positive(H),
        incrBestDo(Expr1,S,H,ERem1,Pol1,V1,Prob1),
        ( ERem1 \== nil,
          ER = (optimize(ERem1) : E),
          Pol = Pol1,             % a policy for on-line execution
          V $= V1, Prob=Prob1 ;
          ERem1 == nil,
          ( Pol1 \== stop -> incrBestDo(Pol1 : E,S,H,ER,Pol,V,Prob) ;
             ( ER = nil, (Prob is 0.0) , Pol = stop, reward(V,S) )
           )
         ).

incrBestDo(nil : E,S,H,ER,Pol,V,Prob) :-  positive(H),
               incrBestDo(E,S,H,ER,Pol,V,Prob).


/* If the program starts with the "stop" action that is not
   possible to execute, then given the reward V in the current
   situation S we solve a linear programming problem VL-> max
   with respect to a system of linear inequalities between temporal
   variables that occur in V. Solving this linear programming problem
   grounds some temporal variables to values of time that provide
   the highest reward with respect to the temporal constraints.
   We seek solutions of similar linear programming problems
   in other branches of the situation tree whenever the directed
   forward search performed by this interpreter gets to a leaf node
   (a final situation).
*/
incrBestDo(stop : E,S,H,ER,Pol,V,Prob) :- positive(H),
                ER=nil, Pol=stop,
                reward(V,S), VL $<= V, rmax(VL),
                (Prob is 0.0) .

/* The program is done, but the horizon is still greater than 0. */
incrBestDo(nil,S,H,ER,Pol,V,Prob) :-   positive(H),
                ER=nil, Pol=nil,
                reward(V,S), VL $<= V, rmax(VL),
                (Prob is 1.0 ) .

/* There is still something to do, but  horizon is already 0. */
incrBestDo(E,S,H,ER,Pol,V,Prob) :-   H = 0,
                ER=nil, Pol=nil,
                reward(V,S), VL $<= V, rmax(VL),
                (Prob is 1.0 ) .
```

```
bestDoAux([N1],E,S,H,Pol,V,Prob) :-   poss(N1,S),
       start(S,T1),  time(N1,T2),  T1 $<= T2,
       prob(N1,Pr1,S),
       incrBestDo(E,do(N1,S),H,_,Pol1,V1,Prob1),
       senseCondition(N1,Phi1),
       Pol = ( ?(Phi1) : Pol1 ),
       rational(Pr1, P_rat),    /* converts Pr1 into the rational number */
       V $= P_rat * V1, /*V1 is linear function with rational coefficients*/
       Prob is Pr1*Prob1.

bestDoAux([N1],E,S,H,Pol,V,Prob) :-   not poss(N1,S),
       senseCondition(N1,Phi1),
       Pol = ( ?(Phi1) : stop ), (Prob is 0.0) , V $= 0 .

bestDoAux([N1 | OtherOutcomes],E,S,H,Pol,V,Prob) :- not OtherOutcomes= [],
       poss(N1,S),
       start(S,T1),  time(N1,T2),  T1 $<= T2,
       bestDoAux(OtherOutcomes,E,S,H,PolTree,VTree,ProbTree),
       incrBestDo(E,do(N1,S),H,_,Pol1,V1,Prob1),
       senseCondition(N1,Phi1),
       Pol = if(Phi1,  % then
                       Pol1,  % else
                                    PolTree),
       prob(N1,Pr1,S),
       rational(Pr1, P_rat),   /* converts Pr1 into the rational number */
       V $= VTree + P_rat*V1,
       Prob is ProbTree + Pr1*Prob1.

bestDoAux([N1 | OtherOutcomes],E,S,H,Pol,V,Prob) :- not OtherOutcomes= [],
       not poss(N1,S), bestDoAux(OtherOutcomes,E,S,H,Pol,V,Prob).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% final(Program,Sit,Hor,Pol,Val) is true iff
% Program cannot be continued or if the computed policy Pol
% cannot be executed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

final(E,S,H,Pol,V) :-  H==0, Pol=nil,
                reward(V,S), VL $<= V, rmax(VL) .

final(nil,S,H,Pol,V) :-  Pol=nil,
                reward(V,S), VL $<= V, rmax(VL) .

final(stop : E,H,S,Pol,V) :-  Pol=stop,
                reward(V,S), VL $<= V, rmax(VL).
```

```
final(E,S,H,Pol,U) :- (Pol == stop ; Pol == nil),
                reward(V,S), VL $<= U, rmax(VL).




    /* ---- Some useful predicates mentioned in the interpreter ----- */

/* Note that evaluation of the predicates "lesseq" and "greatereq"
   can require solving a system of linear inequalities about
   temporal variables.
*/

lesseq(V1,Prob1,V2,Prob2) :-  Pr1 is float(Prob1), Pr2 is float(Prob2),
            (Pr1 \= 0.0 ) , (Pr2 \= 0.0 ) , V1 $<= V2.

lesseq(V1,Prob1,V2,Prob2) :-  Pr1 is float(Prob1), (Pr1 = 0.0 ) ,
         Pr2 is float(Prob2), (Pr2 \= 0.0 ) .

lesseq(V1,Prob1,V2,Prob2) :-  Pr1 is float(Prob1), (Pr1 = 0.0 ) ,
         Pr2 is float(Prob2),  (Pr2 = 0.0 ) ,  V1 $<= V2 .

greatereq(V1,Prob1,V2,Prob2) :- Pr1 is float(Prob1), Pr2 is float(Prob2),
            (Pr1 \= 0.0 ) , (Pr2 \= 0.0 ) , V2 $<= V1.

greatereq(V1,Prob1,V2,Prob2) :-  Pr1 is float(Prob1), Pr2 is float(Prob2),
            (Pr1 \= 0.0 ) , (Pr2 = 0.0 ) .

greatereq(V1,Prob1,V2,Prob2) :-  Pr1 is float(Prob1), Pr2 is float(Prob2),
            (Pr1 = 0.0 ) , (Pr2 = 0.0 ) , V2 $<= V1.


/* In the case of episodic decision task, whenever horizon
   is not known in advance and is determined by transition into a
   terminal (absorbing) state, the value of horizon can be defined by
   the constant "episodic". Note that in this case the predicate
   positive(Horizon) is true.
*/
decrement(Input,Output) :-
      integer(Input), Input > 0, Output is Input - 1.

decrement(Input,Output) :-
      not integer(Arg), Output = Input.

positive(Arg) :-   integer(Arg), Arg > 0.
positive(Arg) :-   not integer(Arg).
```

```
/* A handy primitive action noOp(T) does not change anything. */
agentAction(noOp(T)).
time(noOp(T), T).
poss(noOp(T), S).
deterministic(noOp(T),S).
deterministic(A,S) :- not nondetActions(A,S,OutcomesList).

range([],[]).
range([H | T], [H | T]).


                   /* Time specific clauses.  */

start(do(A,S),T) :- time(A,T).
start( s0, 0 ).

/* Fix on an earliest solution to the temporal constraints.
   This predicate is not used in the current implementation.
*/
chooseTimes(s0).
chooseTimes(do(A,S)) :- chooseTimes(S), time(A,T), T $<= Time, rmin(Time).

/* "now" is a synonym for "start". */
now(T,S) :- start(S,T).


/* The time of an instantenous action is its last argument.
time(X,T) :- X =.. [ActName|Args],
             reverse(Args,[Last | OtherArgs]), Last=T.
*/


/* 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 some
   domain independent atoms, including Prolog system predicates.
   For this to work properly, the GOLOG programmer must provide,
   for those generic atoms taking a situation argument,
   a clause giving the result of restoring its suppressed situation
   argument, for example:
          restoreSitArg(now(Time),S,now(Time,S)).
*/

holds(A,S) :-  restoreSitArg(A,S,F),  F ;
      isAtom(A), not restoreSitArg(A,S,F),   %  calls standard
      A.                                     %  Prolog predicates

isAtom(A) :- not (A = tt ;A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
    A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W)).


/*  It might be convenient to keep the following domain-independent
    clauses in this file.
*/

restoreSitArg(poss(A),S,poss(A,S)).
restoreSitArg(now(T),S,now(T,S)).

/* The representation considered here can be elaborated as follows.
   Modified successor state axioms can be formulated using the
   predicate "ssa". Specifically, ssa-predicates must have 4 arguments:
     - guard condition Alpha ;   % if it is true, then the axiom applies
     - action term  A ;
     - fluent name F ;
     - the right hand side Gamma of the successor state axiom.
```

```
   The domain axiomatization must include the set of unary axioms
   fluent(X)  specifying fluents.
   The domain axiomatization may include several different ssa
   for the same fluent, but their guards must be mutually incompatible.
   If A is a sensing action, then its second before the last argument
   contains data returned from doing sensing in reality.

holds(F,do(A,S)) :- fluent(F),
    ssa(Alpha,A,F,Gamma), holds(Alpha,S), holds(Gamma,S).

holds(tt,S).              % the truth value tt  holds in any situation
holds(F,s0) :- ini(F).    % initially, fluent F is true
holds(-F,s0) :- ini(-F).  % initially, fluent F is false

  Note that in such implementation there would be no need to provide
  restoreSitArg  axioms for fluents:
holds(A,S) :-  restoreSitArg(A,S,F),  F ;
      isAtom(A), not restoreSitArg(A,S,F),   %  calls standard
      not fluent(A), A.                      %  Prolog predicates
*/

/* -----------  cut here  ------------------------ */

stophere(yes).
```

Bibliography

[Ambrose-Ingerson and Steel, 1988] J. Ambrose-Ingerson and S Steel. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 735–740. Morgan Kaufmann, 1988.

[Amir, 2001] Eyal Amir. *Dividing and Conquering Logic*. Ph.D. Thesis, Stanford University, Computer Science Department, 2001.

[Andre and Russell, 2001] David Andre and Stuart Russell. Programmable reinforcement learning agents. In *Proceedings of the 13th Conference on Neural Information Processing Systems (NIPS-01)*, pages 1019–1025. MIT Press, 2001.

[Andre and Russell, 2002] David Andre and Stuart Russell. State abstraction for programmable reinforcement learning agents. In *Proceedings of the 18th National Conference on Artificial Intelligence*, Edmonton, Canada, August 2002. AAAI Press.

[Andre, 2003] David Andre. *Programmable Reinforcement Learning Agents*. Ph.D. Thesis in Computer Science, University of California, Berkeley, CA, 2003.

[Arkin, 1989] R. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, 1989.

[Bacchus and Kabanza, 1996] Fahiem Bacchus and Froduald Kabanza. Using temporal logic to control search in a forward chaining planner. In M. Ghallab and A. Milani, editors, *New Directions in Planning*, pages 141–153, Amsterdam, 1996. IOS Press.

[Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116, 2000.

[Bacchus *et al.*, 1995] Fahiem Bacchus, Joseph Y. Halpern, and Hector J. Levesque. Reasoning about noisy sensors in the situation calculus. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI - 95)*, pages 1933–1940, Montreal, 1995. http://www.cs.toronto.edu/cogrobo/Papers/noise.ps.

[Bacchus *et al.*, 1996] Fahiem Bacchus, Craig Boutilier, and Adam Grove. Rewarding behaviors. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 1160–1167, Portland, OR, 1996.

[Bacchus *et al.*, 1999] F. Bacchus, J.Y. Halpern, and H.J. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence*, 111:171–208, 1999.

[Ballard, 1983] Bruce Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

[Baral and Gelfond, 2000] Chitta Baral and Michael Gelfond. Reasoning agents in dynamic domains. In Jack Minker, editor, *Logic based AI*, pages 257–279. Kluwer Academic Publishers, 2000.

[Baral and Son, 1997] C. Baral and T. Son. Relating theories of actions and reactive control. In *Robots, Softbots, Immobots: Theories of Action, Planning and Control: working notes of the AAAI-97 workshop*, Providence, Rhode Island, 1997.

[Baral and Son, 2001] C. Baral and T. Son. Formalizing sensing actions – a transition function based approach. *Artificial Intelligence*, To appear; available at: http://www.public.asu.edu/ cbaral/, 2001.

[Barruffi *et al.*, 1998] R. Barruffi, E. Lamma, M. Milano, and P Mello. Planning with incomplete and dynamic knowledge via interactive constraint satisfaction. In Ralph Bergmann and Alexander Kott, editors, *AAAI Technical Report WS-98-02, Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments, a workshop held in conjunction with the 4th International Conference on Artificial Intelligence in Planning Systems (available at: http://wwwagr.informatik.uni-kl.de/ bergmann/AIPS98WS/proceedings.html)*, Pittsburgh, Pennsylvania, USA, 1998. AAAI Press.

[Barto and Duff, 1994] A. G. Barto and M. Duff. Monte Carlo matrix inversion and reinforcement learning. In J. D. Cohen, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems: Proceedings of the 1993 Conference*, pages 687–694, San Francisco, CA, 1994. Morgan Kaufmann.

[Barto *et al.*, 1995] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1–2):81–138, 1995.

[Bastié and Régnier, 1996] C. Bastié and P Régnier. Speedy : Monitoring the execution in dynamic environments. In *Reasoning about Actions and Planning in Complex Environments: Proceedings of the Workshop at International Conference on Formal and Applied Practical Reasoning*, Bonn, Germany, 1996.

[Beetz and Bennewitz, 1998] M. Beetz and M. Bennewitz. Planning, scheduling, and plan execution for autonomous robot office couriers. In R. Bergmann and A. Kott, editors, *Proc. of the W/Sh on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments*, Pittsburgh, Pennsylvania, 1998. AAAI Press.

[Beetz and McDermott, 1994] Michael Beetz and Drew McDermott. Improving robot plans during their execution. In Kris Hammond, editor, *Proceedings of the 2nd Int. Conf. on AI Planning Systems (AIPS-94)*, pages 3–12. AAAI Press, 1994.

[Beetz and McDermott, 1996] M. Beetz and D. McDermott. Local planning of ongoing activities. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 19–26. AAAI Press, 1996.

[Beetz and McDermott, 1997] M. Beetz and D. McDermott. Expressing transformations of structured reactive plans. In Sam Steel, editor, *Proceedings of the 4th European Conference on Planning (ECP'97)*, pages 66–78, Toulouse, France, September 24-26, 1997.

[Bellman, 1957] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.

[Benveniste *et al.*, 1990] A. Benveniste, M. Metivier, and P. Priouret. *Adaptive Algorithms and Stochastic Approximations*. Springer Verlag, New York, 1990.

[Bertsekas and Shreve, 1978] Dimitri P. Bertsekas and Steven E. Shreve. *Stochastic optimal control : the discrete time case, ISBN 0120932601*. Academic Press, New York, 1978.

[Bertsekas and Tsitsiklis, 1989] Dimitri P. Bertsekas and John. N. Tsitsiklis. *Parallel and distributed computation : numerical methods*. Prentice Hall, ISBN: 0136487009, Englewood Cliffs, N.J., 1989.

[Bertsekas and Tsitsiklis, 1996] Dimitri P. Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming, ISBN 1886529108*. Athena Scientific, Belmont, Mass., 1996.

[Bjäreland, 2001] Marcus Bjäreland. *Model-based Execution Monitoring*. PhD Thesis, Linköping Studies in Science and Technology, Dissertation No 688, available at http://www.ida.liu.se/labs/kplab/people/marbj/, 2001.

[Blackwell, 1962] David Blackwell. Discrete dynamic programming. *Annals of Mathematical Statistics*, 33(2):719–726, January 1962.

[Blackwell, 1965] David Blackwell. Discounted dynamic programming. *Annals of Mathematical Statistics*, 36(1):226–235, February 1965.

[Bonet and Geffner, 2003] Blai Bonet and Hector Geffner. Labeled rtdp: Improving the convergence of real-time dynamic programming. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS-03)*, Trento, Italy, 2003. ITC-IRST, University of Trento.

[Borgida *et al.*, 1995] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[Borodin and El-Yaniv, 1998] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, 1998.

[Borodin *et al.*, 2002] Allan Borodin, Morten Nielsen, and Charles Rackoff. (Incremental) priority algorithms. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 752–761. Available at: `http://www.cs.toronto.edu/~bor`, 2002.

[Boutilier and Dearden, 1994] Craig Boutilier and Richard Dearden. Using abstractions for decision-theoretic planning with time constraints. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1016–1022, Seattle, 1994.

[Boutilier and Goldszmidt, 1996] Craig Boutilier and Moisés Goldszmidt. The frame problem and Bayesian network action representations. In *Proceedings of the Eleventh Biennial Canadian Conference on Artificial Intelligence*, pages 69–83, Toronto, 1996.

[Boutilier and Puterman, 1995] Craig Boutilier and Martin L. Puterman. Process-oriented planning and average-reward optimality. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1096–1103, Montreal, 1995.

[Boutilier *et al.*, 1995] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1104–1111, Montreal, 1995.

[Boutilier *et al.*, 1999] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[Boutilier *et al.*, 2000a] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level robot programming in the situation calculus. In *Proc. of the 17th National Conference on Artificial Intelligence (AAAI'00)*, pages 355–362, Austin, Texas, 2000. Available at: `http://www.cs.toronto.edu/~cogrobo/`.

[Boutilier *et al.*, 2000b] Craig Boutilier, Richard Dearden, and Moises Goldszmid. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.

[Boutilier *et al.*, 2001] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programming for First-Order MDPs. In Bernhard Nebel, editor, *Proceedings of the 17th International Conference on Artificial Intelligence (IJCAI-01)*, pages 690–700, San Francisco, CA, August, 4–10 2001. Morgan Kaufmann Publishers, Inc.

[Boutilier, 1999] Craig Boutilier. Knowledge representation for stochastic decision processes. In Michael Wooldridge and Manuela Veloso, editors, *Artificial intelligence today: recent trends and developments, Lecture notes in computer science*, volume 1600, pages 111–152. Springer, 1999.

[Bradtke, 1994] S. Bradtke. *Incremental Dynamic Programming for On-line Adaptive Optimal Control*. PhD thesis, University of Massachusetts, 1994.

[Brooks, 1989] Rodney A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. *Neural Computation*, 1(2):253, 1989.

[Burgard *et al.*, 1998] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI'98)*, Madison, Wisconsin, 1998.

[Burgard *et al.*, 1999] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2), 1999.

[Burstall, 1969] R.M. Burstall. Formal description of program structure and semantics in first order logic. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 79–98. Edinburgh University Press, 1969.

[Chang and Marcus, 2003] Hyeong Soo Chang and Steven I. Marcus. Approximate receding horizon approach for markov decision processes: average reward case. *Journal of Mathematical Analysis and Applications*, 286(2):636–651; available at: http://techreports.isr.umd.edu/ARCHIVE/, October 2003.

[Cimatti and Roveri, 2000] Alessandro Cimatti and Marco Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338, December 2000.

[Çinlar, 1975] Erhan Çinlar. *Introduction to stochastic processes*. Prentice-Hall, 1975.

[Cohen *et al.*, May 1997] Don Cohen, Martin S. Feather, K. Narayanaswamy, and Stephen S. Fickas. Automatic monitoring of software requirements. In *Proceedings of the 19th International Conference on Software Engineering (ICSE)*, pages 602–603, Boston, MA, May 1997. ACM Press.

[Coradeschi and Saffiotti, 2000] S. Coradeschi and A. Saffiotti. Anchoring symbols to sensor data: preliminary report. In *Proc. of the 17th AAAI Conf.*, pages 129–135, Menlo Park, CA, 2000. AAAI Press. Online at http://www.aass.oru.se/˜asaffio/.

[Coradeschi, 1996] Silvia Coradeschi. Reasoning with misperception in the features and fluents framework. In *Proc. of the 12th European Conference on Artificial Intelligence (ECAI-96)*, 12-16 August, Budapest, Hungary, 1996.

[Dantzig and Wolfe, 1960] George Dantzig and Philip Wolfe. Decomposition principle for dynamic programs. *Operations Research*, 8(1):101–111, 1960.

[Davis and Morgenstern, 1993a] Ernest Davis and Leora Morgenstern. Epistemic logics and their applications. Technical report, IJCAI-93 tutorial, available at: http://www-formal.stanford.edu/leora/krcourse/, 1993.

[Davis and Morgenstern, 1993b] Ernest Davis and Leora Morgenstern. Epistemic logics: Annotated bibliography. Technical report, IJCAI-93 tutorial on Epistemic Logics and their Applications; available at: http://www-formal.stanford.edu/leora/krcourse/, 1993.

[De Giacomo and Levesque, 1999a] G. De Giacomo and H. Levesque. Projection using regression and sensors. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, 1999.

[De Giacomo and Levesque, 1999b] G. De Giacomo and H.J. Levesque. An incremental interpreter for high-level programs with sensing. In Hector Levesque and Fiora Pirri, editors, *Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages 86–102. Springer, 1999.

[De Giacomo *et al.*, 1997a] G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati. Planning with sensing for a mobile robot. In *Proc. of the 4th European Conference on Planning (ECP'97)*, pages 158–170. Springer Verlag, LNCS 1348, 1997.

[De Giacomo *et al.*, 1997b] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent executions, prioritized interrupts, and exogenous actions in the situation calculus. In *15th International Joint Conference on Artificial Intelligence (IJCAI–97)*, volume 2, pages 1221–1226, Nagoya, Japan, 1997.

[De Giacomo *et al.*, 1998] G. De Giacomo, R. Reiter, and M.E. Soutchanski. Execution monitoring of high-level robot programs. In *Principles of Knowledge Representation and Reasoning: Proc. of the 6th International Conference (KR'98)*, pages 453–464, Trento, Italy, 1998.

[De Giacomo *et al.*, 1999] G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati. A theory and implementation for mobile robot agents. *Journal of Logic and Computation*, 9(5), 1999.

[De Giacomo *et al.*, 2000] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[De Giacomo *et al.*, 2001] Giuseppe De Giacomo, Hector J. Levesque, and Sebastian Sardiña. Incremental execution of guarded theories. *ACM Transactions on Computational Logic (TOCL)*, 2(4):495–525, October 2001.

[De Giacomo *et al.*, 2002] Giuseppe De Giacomo, Yves Lespérance, Hector Levesque, and Sebastian Sardiña. On the semantics of deliberation in IndiGolog – from theory to implementation. In D. Fensel, F. Giunchiglia, D. McGuinness, and M. A. Williams, editors, *Proceedings of Eighth International Conference in Principles of Knowledge Representation and Reasoning (KR-2002)*, pages 603–614, Toulouse, France, April 2002. Morgan Kaufmann.

[Dean and Kanazawa, 1989] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[Dean and Lin, 1995a] Thomas Dean and Shieu-Hong Lin. Decomposition techniques for planning in stochastic domains. In *IJCAI'95, Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1121–1127, Montreal, Canada, 1995. Morgan Kaufmann.

[Dean and Lin, 1995b] Thomas Dean and Shieu-Hong Lin. Decomposition techniques for planning in stochastic domains. Technical report, Technical Report CS-95-10, Dept. of Computer Science, Brown University, available at: http://www.cs.brown.edu/publications/techreports/reports/CS-95-10.html, 1995.

[Dean *et al.*, 1995] T. Dean, L. Kaelbling, J. Kirman, and Nicholson A. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76, N. 1-2:35–74, 1995.

[Dearden and Boutilier, 1994] Richard Dearden and Craig Boutilier. Integrating planning and execution in stochastic domains. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 162–169, Washington, DC, 1994.

[Dearden and Boutilier, 1997] Richard Dearden and Craig Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 89:219–283, 1997.

[Demolombe and Pozos-Parra, 2000] Robert Demolombe and Maria Pozos-Parra. A simple and tractable extension of situation calculus to epistemic logic. In Setsuo Ohsuga and Zbigniew Ras, editors, *Twelfth International Symposium on Methodologies for Intelligent Systems (ISMIS-00)*, pages 515–524, Charlotte, NC, October 11-14, 2000. Springer Verlag, LNAI, v.1932.

[Derman, 1970] Cyrus Derman. *Finite State Markovian Decision Processes*. New York, Academic Press, 1970.

[Dietterich, 2000] Thomas Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[Doherty, 1999] Patrick Doherty. The WITAS integrated software system architecture. *Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, available at: http://www.ep.liu.se/ea/cis/1999/017/*, 4(17), 1999.

[Dynkin and Yushkevich, 1979] E.B. Dynkin and A.A. Yushkevich. *Controlled Markov processes [Upravl?i'aemye markovskie pro?t'sessy i ikh prilozheni?i'a]*. Springer-Verlag, Berlin ; New York, 1979.

[Earl and Firby, 1996] C. Earl and R.J. Firby. Combined execution and monitoring for control of autonomous agents. Technical report, University of Chicago, TR-96-19. Available at: http://cs-www.uchicago.edu/publications/tech-reports/, 1996.

[Enderton, 2001] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Harcourt Press, Second edition, 2001.

[Feather *et al.*, April 1998] M.S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behaviour. In *Proceedings de IWSSD'98 - 9th International Workshop on Software Specification and Design*, Isobe, Japan, April 1998. IEEE.

[Fel'dbaum, 1965] A.A. Fel'dbaum. *Optimal control systems*. Academic Press, New York, 1965. Translated from Russian by A. Kraiman.

[Feldman and Harel, 1984] Yishai Feldman and David Harel. A probabilistic dynamic logic. *Journal of Computer and System Sciences*, 28:193–215, 1984.

[Feng *et al.*, 2003] Zhengzhu Feng, Eric A. Hansen, and Shlomo Zilberstein. Symbolic generalization for on-line planning. In *The Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI-03)*, pages 109–116, Acapulco, Mexico, 2003.

[Ferrein *et al.*, 2003] Alexander Ferrein, Christian Fritz, and Gerhard Lakemeyer. Extending DTGolog with Options. In *Proc of the 18th International Joint Conference on Artificial Intelligence*, 2003.

[Ferrein *et al.*, submitted] A. Ferrein, C. Fritz, and G. Lakemeyer. On-line decision-theoretic golog for unpredictable domains. submitted. Available at: http://www-kbsg.informatik.rwth-aachen.de/staff/publications.php?user_ID=1.

[Fikes *et al.*, 1972] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Learning and executing generalized robot plans. Artificial Intelligence, 3(4):251–288, 1972.

[Fine, 1973] Terrence Fine. *Theories of probability; an examination of foundations*. Academic Press, New York, 1973.

[Finzi *et al.*, 2000] Alberto Finzi, Fiora Pirri, and Ray Reiter. Open world planning 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 754–760, Menlo Park, CA, July 30– 3 2000. AAAI Press.

[Firby, 1989] R.J. Firby. *Adaptive Execution in Complex Dynamic Worlds, Ph.D. Thesis*. Dept. of Computer Science, Yale University Report RR#672, 1989.

[Fritz, 2003] Christian Fritz. *Integrating decision-theoretic planning and programming for robot control in highly dynamic domains*. RWTH Aachen, Germany, 2003.

[Funge, 1998] J. Funge. *Making Them Behave: Cognitive Models for Computer Animation, Ph.D. Thesis*. Dept. of Computer Science, Univ. of Toronto, 1998.

[Gabaldon, 2003] Alfredo Gabaldon. Compiling control knowledge into preconditions for planning in the situation calculus. In *Proc. of the 18th International Joint Conference on Artificial Intelligence, IJCAI-03*, Acapulco, Mexico, August 9-15, 2003.

[Gat, 1996] E. Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In *AAAI-96 Fall Symposium on Plan Execution*, Boston, MA, 1996. AAAI.

[Genesereth and Nourbakhsh, 1993] Michael Genesereth and Illah Nourbakhsh. Time-saving tips for problem-solving with incomplete information. In *Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI-93*, 1993.

[Georgeff and Ingrand, 1989] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 972–978, Detroit, MI, 1989.

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

[Giunchiglia *et al.*, 1994] F. Giunchiglia, P. Traverso, and L. Spalazzi. Planning with failure. In *2nd International Conference on AI Planning Systems (AIPS-94)*, Chicago, IL, June 15-17, 1994.

[Golden and Weld, 1996] Keith Golden and Daniel Weld. Representing sensing actions: The middle ground revisited. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pages 174–185. Morgan Kaufmann, San Francisco, California, 1996.

[Golden *et al.*, February 1996] K. Golden, O. Etzioni, and D. Weld. Planning with execution and incomplete information. Technical report, Technical Report UW-CSE-96-01-09, Dept. of Computer Science and Engineering, University of Washington. Available at: http://www.cs.washington.edu/homes/weld/pubs.html, February 1996.

[Goldman and Boddy, 1996] R.P. Goldman and M. Boddy. Expressive planning and explicit knowledge. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 110–117. AAAI Press, 1996.

[Green, 1969] C.C. Green. Theorem proving by resolution as a basis for question–answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 343–348, 1969.

[Green, 1980] Claude Cordell Green. *The application of theorem proving to question-answering systems*. Garland Pub., Originally presented in 1969 as the author's PhD thesis in the computer science, Stanford University, New York, 1980.

[Grosskreutz and Lakemeyer, 2001] Henrik Grosskreutz and Gerhard Lakemeyer. Belief update in the pgolog framework. In *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001), Lecture Notes on Artificial Intelligence*, volume 2174, pages 213–228, September 19-21, 2001, Vienna, Austria, 2001. Springer Verlag.

[Grosskreutz, 2000] H. Grosskreutz. Probabilistic projection and belief update in the pgolog framework. In *The 2nd International Cognitive Robotics Workshop, 14th European Conference on AI*, pages 34–41, Berlin, Germany, 2000.

[Grosskreutz, 2002] Henrik Grosskreutz. *Towards More Realistic Logic-Based Robot Controllers in the GOLOG Framework*. RWTH Aachen, Germany, 2002.

[Gu, 2002] Yilan Gu. *handling uncertainty systems in the situation calculus with macro actions*. University of Toronto, Department of Computer Science, Master of Science thesis, 2002.

[Hähnel *et al.*, 1998] D. Hähnel, W. Burgard, and G. Lakemeyer. Golex - bridging the gap between logic (GOLOG) and a real robot. In *Proceedings of the 22nd German Conference on Artificial Intelligence (KI 98)*, Bremen, Germany, 1998.

[Hähnel, 1998] D. Hähnel. *GOLEX: Ein Laufzeitsystem für die Aktionsbeschreibungssprache GOLOG zur Steuerung des mobilen Roboters RHINO*. Diplomarbeit an der Universität Bonn. Available at: http://titan.informatik.uni-bonn.de/˜haehneld/, 1998.

[Haigh and Veloso, 1996] K.Z. Haigh and M. Veloso. Interleaving planning and robot execution for asynchronous user requests. In *International Conf. on Intelligent Robots and Systems (IROS-96)*, pages 148–155, Osaka, Japan, 1996.

[Hanks and McDermott, 1994] Steve Hanks and Drew V. McDermott. Modeling a dynamic and uncertain world i: Symbolic and probabilistic reasoning about change. *Artificial Intelligence*, 1994.

[Hansen and Cohen, 1992] E.A. Hansen and P.R. Cohen. Learning a decision rule for monitoring tasks with deadlines. Technical report, 92-80, Experimental Knowledge Systems Lab., Dept. of Computer Science, University of Massachusetts/Amherst. Available at: ftp.cs.umass.edu/pub/eksl/tech-reports/92-80.ps, 1992.

[Harel *et al.*, 2000] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, ISBN:0262082896, 2000.

[Hauskrecht *et al.*, 1998] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 220–229, Madison, WI, 1998.

[Hernández-Lerma and Lasserre, 1990] O Hernández-Lerma and J.B. Lasserre. Error bounds for rolling horizon policies in general markov control processes. *IEEE Transactions on Automatic Control*, 35:1118–1124, 1990.

[Hernández-Lerma and Lasserre, 1996] O Hernández-Lerma and J.B. Lasserre. *Discrete-time Markov control processes : basic optimality criteria*. Springer-Verlag, ISBN 0387945792, New York, 1996.

[Hinderer, 1970] K. Hinderer. *Foundations of non-stationary dynamic programming with discrete time parameter*. Springer-Verlag, Berlin, 1970.

[Hintikka, 1962] Jaakko Hintikka. *Knowledge and belief: an introduction to the logic of the two notions*. Ithaca, N.Y.: Cornell University Press, 1962.

[Hoey *et al.*, 1999] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288, Stockholm, 1999.

[Hoey *et al.*, 2000] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. Optimal and approximate stochastic planning using decision diagrams. Technical report, University of British Columbia Technical Report TR-00-05, available at: http://www.cs.ubc.ca/spider/staubin/Spudd/, 2000.

[Horty and Pollack, 1998] J. Horty and M.E. Pollack. Plan management issues for cognitive robotics: Project overview. In *Proceedings of the 1998 AAAI Fall Symposium on Cognitive Robotics*, Orlando, FL, 1998. http://bert.cs.pitt.edu/˜pollack/distrib/chrono-pubs.html.

[Howard, 1960] Ronald Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[Howard, 1971] Ronald Howard. *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes*. Wiley, N.Y., 1971.

[Ishida, 1997] T. Ishida. *Real-time Search for Learning Autonomous Agents*. Kluwer Academic Publishers, 1997.

[Jaakkola *et al.*, 1994] T Jaakkola, M. Jordan, and S. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201, 1994.

[Kaelbling *et al.*, 1996] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[Kearns *et al.*, 1999] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, Stockholm, 1999.

[Kibler and Morris, 1981] Dennis F. Kibler and Paul Morris. Do not be stupid. In *Proc. of the 7th International Joint Conference on Artificial Intelligence, IJCAI-1981*, pages 345–347, Vancouver, BC, Canada, 1981.

[Koenig and Simmons, 1995] S. Koenig and R. Simmons. Real-time search in nondeterministic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1660–1667, Montreal, 1995.

[Koenig, 2001] Sven Koenig. Minimax real-time heuristic search. *Artificial Intelligence*, 129(1-2):165–197, 2001.

[Kohn and Nerode, 1993a] W. Kohn and A. Nerode. Autonomous control of hybrid systems with declarative controllers. In *Proc. of the 2nd International Workshop on Logic Programming and Non-Monotonic Reasoning, LPNMR-1993*, pages 3–22, Lisbon, Portugal, 1993. The MIT Press.

[Kohn and Nerode, 1993b] W. Kohn and A. Nerode. Multiple agents autonomous control: A hybrid systems architecture. In *Logical methods: in honor of Anil Nerode's 60th birthday*, pages 593–623, Boston, 1993. Birkhauser.

[Kohn, 1988] W. Kohn. A declarative theory for rational controllers. In *Proc. of the 27th IEEE Conference on Decision and Control*, pages 130–136, Austin, Texas, 1988. IEEE Press.

[Kohn, 1991] W. Kohn. Declarative control architecture. *Communications of the ACM*, 34(18):35–46, 1991.

[Kolmogorov and Fomin, 1970] A.N. Kolmogorov and S.V. Fomin. *Introductory Real Analysis*. Dover Publications, New York, 1970.

[Kolmogorov, 1983] A.N. Kolmogorov. On logical foundations of probability theory. In K. Ito and Y.V.Prokhorov, editors, *Probability Theory and Mathematical Statistics, Proceedings of the 4th USSR-Japan Symposium held in Tbilisi, August 23-29, 1982, Lecture Notes in Mathematics*, volume 1021, pages 1–5, Berlin, 1983. Springer Verlag.

[Konolige, 1982] Kurt Konolige. A first order formalization of knowledge and action for a multi-agent planning system. In J. Hays and D. Michie, editors, *Machine Intelligence*, volume 10, Chichester, UK, 1982. Ellis Horwood.

[Korf, 1985a] Richard Korf. *Learning To Solve Problems by Searching for Macro-Operators*. Pitman Publishing Ltd, 1985.

[Korf, 1985b] Richard Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[Korf, 1990] Richard Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[Korolyuk and Korolyuk, 1999] Vladimir S. Korolyuk and Vladimir V. Korolyuk. *Stochastic models of systems*. Kluwer Academic, 1999.

[Kozen, 1981] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.

[Kozen, 1985] Dexter Kozen. A probabilistic PDL. *Journal of Computer and System Sciences*, 30:162–178, 1985.

[Kripke, 1963a] S. Kripke. A semantical analysis of modal logic i: normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen Mathematik*, 9:67–96, 1963.

[Kripke, 1963b] S. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963. Also reprinted in Linsky, Leonard (ed.), *Reference and modality*, Oxford, UK: Oxford University Press, 1971, pp. 63–72.

[Kushmerick *et al.*, 1995] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76:239–286, 1995.

[Kvarnström and Doherty, 2001] J. Kvarnström and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2001.

[Laird and Rosenbloom, 1990] John E. Laird and Paul Rosenbloom. Integrating execution, planning, and learning in Soar for external environment. In *Proceedings of National Conference of Artificial Intelligence*, Boston, MA, 1990.

[Lakemeyer and Levesque, 1998] G. Lakemeyer and H.J. Levesque. AOL: a logic of acting, sensing, knowing, and only knowing. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pages 316–327, 1998.

[Lakemeyer, 1999] G. Lakemeyer. On sensing and off-line interpreting in golog. In Levesque and Pirri, editors, *Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages 173–189. Springer, 1999.

[Lamsweerde and Letier, 2000] Axel Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, 26(10):978–1005, 2000.

[Lespérance and Levesque, 1995] Y. Lespérance and H.J. Levesque. Indexical knowledge and robot action - a logical account. *Artificial Intelligence*, 73:69–115, 1995.

[Lespérance and Ng, 2000] Y. Lespérance and H.-K. Ng. Integrating planning into reactive high-level robot programs. In *Proceedings of the Second International Cognitive Robotics Workshop held in conjunction with ECAI-2000*, pages 49–54, Berlin, Germany, August 21-22, 2000.

[Lespérance *et al.*, 1998] Y. Lespérance, K. Tam, and M. Jenkin. Reactivity in a logic-based robot programming framework. In *1998 AAAI Fall Symposium on Cognitive Robotics, Technical Report FS-98-02*, pages 98–105, Orlando, Florida, USA, 1998. AAAI Press.

[Lespérance, 1991] Yves Lespérance. *A Formal Theory of Indexical Knowledge and Action*. Ph.D. Thesis, Department of Computer Science, University of Toronto, 1991.

[Letier and van Lamsweerde, 2002] Emmanuel Letier and Axel van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'2002)*, Orlando, Florida, USA, May, 2002.

[Levesque and Lakemeyer, 2000] Hector Levesque and Gerhard Lakemeyer. *The Logic of Knowledge Bases*. MIT Press, 2000.

[Levesque and Pagnucco, 2000] H.J. Levesque and M. Pagnucco. Legolog: Inexpensive experiments in cognitive robotics. In *The 2nd International Cognitive Robotics Workshop (held in conjunction with ECAI-2000)*, pages 104–109, Berlin, Germany, 2000.

[Levesque and Reiter, 1998] Hector Levesque and Ray Reiter. High-level robotic control: Beyond planning. a position paper. In *AIII 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap*, March 1998.

[Levesque *et al.*, 1997] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog : A logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31, N 1–3:59–83, 1997.

[Levesque *et al.*, 1998] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for a calculus of situations. *Electronic Transactions of AI (ETAI)*, 2(3–4):159–178, 1998.

[Levesque, 1996] H.J. Levesque. What is planning in the presence of sensing? In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, volume 2, pages 1139–1145, Portland, Oregon, 1996.

[Li and Vitanyi, 1997] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, ISBN 0-387-94868-6, 1997.

[Lin and Reiter, 1997a] Fangzhen Lin and Ray Reiter. How to progress a database. *Artificial Intelligence*, 92:131–167, 1997.

[Lin and Reiter, 1997b] Fangzhen Lin and Ray Reiter. Rules as actions: A situation calculus semantics for logic programs. *Journal of Logic Programming, Special issue on Reasoning about Action and Change*, 31:299–330, 1997.

[Lin, 1997] Fangzhen Lin. Applications of the situation calculus to formalizing control and strategic information: The Prolog cut operator. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1412–1418, 1997. (IJCAI-97 Distinguished Paper Award).

[Littman *et al.*, 1995] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 394–402, Montreal, 1995.

[Littman *et al.*, 1998] M.L. Littman, J. Goldsmith, and M. Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9:1–36, 1998.

[Littman, 1997] Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 748–754, Providence, RI, 1997.

[Manna and Waldinger, 1980] Zohar Manna and Richard J. Waldinger. Problematic features of programming languages: a situational-calculus approach. Technical report, Stanford University, Department of Computer Science, 1980. Report Number: CS-TR-80-779, available at http://www-db.stanford.edu/TR/CS-TR-80-779.html.

[Manna and Waldinger, 1981] Zohar Manna and Richard J. Waldinger. Problematic features of programming languages: a situational-calculus approach. *Acta Informatica*, 16:371–426, 1981.

[Mayne *et al.*, 2000] D.Q. Mayne, J.B. Rawlings, C.V. Rao, and P.O.M. Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36, Issue 6:789–814, 2000.

[McAllester, 2000] David McAllester. Bellman equations for stochastic programs, 2000. http://ttic.uchicago.edu/ dmcallester/.

[McCarthy and Hayes, 1969] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969.

[McCarthy, 1963] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.

[McCarthy, 1990] John McCarthy. *Formalization of common sense: papers by John McCarthy edited by V. Lifschitz*. Ablex, Norwood, N.J., 1990.

[McFarland and Bösser, 1993] D. McFarland and T. Bösser. *Intelligent Behavior in Animals and Robots*. A Bradford Book, The MIT Press, 1993.

[McIlraith, 1997] S. McIlraith. *Towards a Formal Account of Diagnostic Problem Solving*. Ph.D. Thesis, Department of Computer Science, University of Toronto, 1997.

[McIlraith, 1998] S. McIlraith. Explanatory diagnosis: Conjecturing actions to explain obsevations. In *Principles of Knowledge Representation and Reasoning: Proc. of the 6th International Conference (KR'98)*, pages 167–177, Italy, 1998.

[McIlraith, 2000] S. McIlraith. Integrating actions and state constraints: A closed-form solution to the ramification problem (sometimes). *Artificial Intelligence*, 116:87–121, 2000.

[Minton, 1988] Steven Minton. *Learning Search Control Knowledge*. Kluwer Academic Publishers, 1988.

[Moore, 1985] R.C. Moore. A formal theory of knowledge and action. In *Formal Theories of the Commonsense World*, pages 319–358. Ablex, 1985.

[Morgenstern, 1988] Leora Morgenstern. *Foundations of a Logic of Knowledge, Action, and Communication*. Ph.D. Thesis, Department of Computer Science, New York University, 1988.

[Mundhenk *et al.*, 2000] Martin Mundhenk, Judy Goldsmith, Christopher Lusena, and Eric Allender. Complexity of finite-horizon markov decision process problems. *Journal of ACM*, 47(4):681–720, 2000.

[Musliner *et al.*, 1991] D.J. Musliner, E.H. Durfee, and K.G. Shin. Execution monitoring and recovery planning with time. In *Proc. Conf. on Artificial Intelligence Applications. Available at: http://www.cs.umd.edu/users/musliner/*, 1991.

[Musliner *et al.*, 1995] D.J. Musliner, E.H. Durfee, and K.G. Shin. World modeling for the dynamic construction of real-time control plans. *Artificial Intelligence*, 74, N 1:83–127, 1995.

[Nakamura *et al.*, 2000] M. Nakamura, C. Baral, and M. Bjreland. Maintainability: a weaker stabilizability-like notion for high level control of agents. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI '00)*, pages 62–67, Austin, Texas, USA, 2000.

[Nilsson, 1969] Nils J. Nilsson. A mobile automaton: An application of artificial intelligence techniques. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 509–520, Washigton, D.C., 1969.

[Nourbakhsh, 1997] I.R. Nourbakhsh. *Interleaving Planning and Execution, Ph.D. Thesis*. Dept. of Computer Science, Stanford University, Report N STAN-CS-TR-97-1593, 1997.

[Papadimitriou and Tsitsiklis, 1987] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of markov decision processes. *Mathematics of Operations Research*, 12(3):441 – 450, 1987.

[Parr and Russell, 1998] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In M. Kearns M. Jordan and S. Solla, editors, *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT Press, Cambridge, 1998.

[Parr, 1998a] Ronald Parr. Flexible decomposition algorithms for weakly coupled Markov decision processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 422–430, Madison, WI, 1998.

[Parr, 1998b] Ronald Parr. *Hierarchical Control and Learning in Markov Decision Processes*. Ph.D. Thesis in Computer Science, University of California, Berkeley, CA, 1998.

[Pednault, 1986] E.P.D. Pednault. *Towards a Mathematical Theory of Plan Synthesis, Ph.D. Thesis*. Department of Electrical Engineering, Stanford University, 1986.

[Petrick and Levesque, 2002] Ron Petrick and Hector Levesque. Knowledge equivalence in combined action theories. In *Proceedings of KR-2002*, Toulouse, France, April 2002. To appear.

[Pfeffer, 2001] Avi Pfeffer. IBAL: A probabilistic rational programming language. In *Proceedings of IJCAI-2001*, Seattle, WA, USA, August 2001.

[Piazzolla *et al.*, 2000] G. Piazzolla, M. Vaccaro, A. Finzi, and F. Pirri. Armhand: a mobile manipulator for the blocks world. In *The 2nd International Cognitive Robotics Workshop (held in conjunction with ECAI-2000)*, pages 110–116, Berlin, Germany, 2000.

[Pirri and Finzi, 1999] F. Pirri and A. Finzi. An approach to perception in theory of actions: part 1. *Linköping Electronic Articles in Computer and Information Science*, 4(41), 1999.

[Pirri and Reiter, 1999] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):261–325, 1999.

[Poole, 1997] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):7–56, 1997.

[Poole, 1998] D. Poole. Decision theory, the situation calculus and conditional plans. *Linköping Electronic Articles in Computer and Information Science. Available at: http://www.ep.liu.se/ea/cis/1998/008/*, 3: nr 008, 1998.

[Precup and Sutton, 1998] Doina Precup and Richard S. Sutton. Multi-time models for temporally abstract planning. In M. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 10 (Proceedings of NIPS'97)*, pages 1050–1056. MIT Press, Cambridge, 1998.

[Precup *et al.*, 1998] Doina Precup, Richard S. Sutton, and Satinder Singh. Theoretical results on reinforcement learning with temporally abstract behaviors. In *Proceedings of the Tenth European Conference on Machine Learning*, pages 382–393, Chemnitz, Germany, 1998.

[Precup, 2000] Doina Precup. *Temporal Abstraction in Reinforcement Learning*. PhD dissertation, University of Massachusetts, Amherst, 2000.

[Puterman, 1994] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

[Raiffa, 1968] Howard Raiffa. *Decision analysis: introductory lectures on choices under uncertainty*. Addison-Wesley, Reading (MA), 1968.

[Ramadge and Wonham, 1989] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[Reiter, 1991] Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy)*, pages 359–380. Academic Press, San Diego, 1991.

[Reiter, 1998] R. Reiter. Sequential, temporal golog. In A.G. Cohn and L.K. Schubert, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the 6th International Conference (KR'98)*, pages 547–556, Trento, Italy, 1998. Morgan Kaufmann Publishers, San Francisco, CA.

[Reiter, 2001a] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, 2001.

[Reiter, 2001b] Ray Reiter. On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic*, 2(4):433–457, October 2001.

[Robbins and Monro, 1951] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[Ross, 1983] Sheldon Ross. *Introduction to stochastic dynamic programming*. Academic Press, New York, 1983.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Sandewall, 1995] Erik Sandewall. *Features and Fluents: The Representation of Knowledge About Dynamical Systems*. Oxford University Press, 352 pages, 1995.

[Sandewall, 1997] E Sandewall. Logic-based modelling of goal-directed behavior. *Linköping Electronic Articles in Computer and Information Science*, 2(19), 1997.

[Sandewall, 1998] Erik Sandewall. Cognitive robotics logic and its metatheory: Features and fluents revisited. *Linköping Electronic Articles in Computer and Information Science*, 3(17):http://www.ep.liu.se/ea/cis/1998/017/, 1998.

[Sardina *et al.*, submitted] Sebastian Sardina, Yves De Giacomo, Giuseppe Lespénce, and Hector Levesque. On the semantics of deliberation in IndiGolog – from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, submitted. Previous version appeared in Proc. of KR-2002.

[Scherl and Levesque, 1993] R. Scherl and H.J. Levesque. The frame problem and knowledge producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 689–695, Washington, DC, 1993.

[Scherl and Levesque, 2003] Richard B. Scherl and Hector J. Levesque. Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1-2):1–39, 2003.

[Schoppers, 1987] M.J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, Milan, Italy, 1987. Morgan Kaufmann Publishers.

[Schoppers, 1989] M.J. Schoppers. In defense of reaction plans as caches. *Artificial Intelligence Magazine*, 10(4):51–60, 1989.

[Schoppers, 1992] M.J. Schoppers. Building monitors to exploit open-loop and closed-loop dynamics. In *Proceedings of the 1st International Conference on Artificial Iintelligence Planning Systems (AIPS-92)*, pages 204–213, College Park, Maryland, June 15-17, 1992. Morgan Kaufmann Publishers.

[Shanahan, 1997] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. The MIT Press, 410 pages, 1997.

[Shanahan, 1998] M.P. Shanahan. A logical account of the common sense informatic situation for a mobile robot. *Electronic Transactions on Artificial Intelligence*, 2:69–104; available at: http://www–ics.ee.ic.ac.uk/ mpsha/pubs.html, 1998.

[Shapiro *et al.*, 2000] S. Shapiro, M. Pagnucco, Y. Lespérance, and H.J. Levesque. Iterated belief change in the situation calculus. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning*, Breckenridge, Colorado, 2000. Morgan Kaufmann Publishers, San Francisco, CA.

[Sierra-Santibáñez, 2001] Josefina Sierra-Santibáñez. Heuristic planning: a declarative forward chaining approach. In *The 5th Symposium on Logical Formalizations of Commonsense Reasoning, available at: http://www.cs.nyu.edu/faculty/davise/commonsense01/papers.html*, Courant Institute of mathematical Sciences, New York University, New York, USA, May 20-22, 2001.

[Singh and Yee, 1994] S. Singh and R.C. Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16:227–233, 1994.

[Singh, 1992a] Satinder P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proc. of the 10th National Conference on Artificial Intelligence*, pages 202–207, San Jose, CA, 1992.

[Singh, 1992b] Satinder P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3):323–339, 1992.

[Singh, 1993] Satinder Singh. *Learning to Solve Markovian Decision Tasks*. Ph.D. Thesis, University of Massachussets, Amherst, MA, 1993.

[Soutchanski, 1999a] Mikhail Soutchanski. Execution monitoring of high–level temporal programs. In Michael Beetz and Joachim Hertzberg, editors, *Robot Action Planning, Proceedings of the IJCAI-99 Workshop*, pages 47–54, Stockholm, Sweden, 1999. available at `http://www.cs.toronto.edu/~mes/papers`.

[Soutchanski, 1999b] Mikhail Soutchanski. Execution monitoring of high–level temporal programs. In Abdel-Illah Mouaddib and Thierry Vidal, editors, *Scheduling and Planning Meet Real-Time Monitoring in a Dynamic and Uncertain World, proceedings of the IJCAI-99 Workshop*, pages 71–78, Stockholm, Sweden, 1999. available at `http://www.cs.toronto.edu/~mes/papers`.

[Soutchanski, 2000] Mikhail Soutchanski. An on-line decision-theoretic golog interpreter. In Gerhard Lakemeyer, editor, *The Second International Cognitive Robotics Workshop held in conjunction with 14th European Conference on Artificial Intelligence ECAI-2000*, pages 117–124, Berlin, Germany, August 21-22, 2000. available at `http://www.cs.toronto.edu/~mes/papers`.

[Soutchanski, 2001a] Mikhail Soutchanski. A correspondence between two different solutions to the projection task with sensing. In *The 5th Symposium on Logical Formalizations of Commonsense Reasoning*, pages 235–242, Courant Institute of mathematical Sciences, New York University, New York, USA, May 20-22, 2001.

[Soutchanski, 2001b] Mikhail Soutchanski. An on-line decision-theoretic golog interpreter. In *17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 19–24, Seattle, Washington, USA, 2001.

[Stone and Veloso, 1999] Peter Stone and Manuela Veloso. User-guided interleaving of planning and execution. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*. IOS Press, 1999.

[Sutton and Barto, 1998a] Richard S Sutton and Andrew G Barto. *Reinforcement learning: an introduction*. MIT Press, Cambridge, Mass., 1998.

[Sutton and Barto, 1998b] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[Sutton *et al.*, 1999] Richard Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[Sutton, 1988] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[Sutton, 1990] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

[Sutton, 1995] Richard S. Sutton. TD models: Modeling the world at a mixture of time scales. In A. Prieditis and S. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 531–539, Lake Tahoe, Nevada, 1995.

[Thielscher, 2000a] Michael Thielscher. *Challenges for Action Theories*. Springer Verlag, 2000.

[Thielscher, 2000b] Michael Thielscher. Representing the knowledge of a robot. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 109–120, Breckenridge, CO, April 2000. Morgan Kaufmann.

[Thrun *et al.*, 1999] S. Thrun, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. MINERVA: A second generation mobile tour-guide robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1999.

[Thrun, 1997] S. Thrun. To know or not to know: The role of models in mobile robotics. *AI Magazine*, 18, N 1, 1997.

[Thrun, 2000] S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA, 2000. IEEE. http://robots.stanford.edu/papers/thrun.ces-icra.html.

[Traverso and Spalazzi, 1995] P. Traverso and L. Spalazzi. A logic for acting, sensing and planning. In *14th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 1941–1949, Montreal, Canada, 1995.

[Traverso *et al.*, 1992] P. Traverso, A. Cimatti, and L. Spalazzi. Beyond the single planning paradigm: introspective planning. In *10th European Conference on Artificial Intelligence (ECAI-92)*, pages 643–647, Vienna, Austria, 1992.

[Tsitsiklis, 2002] John Tsitsiklis. On the convergence of optimistic policy iteration. *Journal of Machine Learning Research*, 3:59–72, 2002.

[Tsypkin, 1971] Ya. Z. Tsypkin. *Adaptation and learning in automatic systems*. Academic Press, 1971.

[Veloso *et al.*, 1998] M.M. Veloso, M.E. Pollack, and M.T. Cox. Rationale-based monitoring for planning in dynamic environments. In *Proc. of the 4th International Conference on AI Planning Systems*, pages 171–179, Pittsburgh, PA, 1998.

[Waldinger, 1977] R. Waldinger. Achieving several goals simultaneously. In E. Elcock and D. Michie, editors, *Machine Intelligence*, volume 8, pages 94–136, Edinburgh, Scotland, 1977. Ellis Horwood.

[Watkins and Dayan, 1992] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[Watkins, 1989] C.J.C.H. Watkins. *Learning from delayed rewards*. Ph.D. Thesis, University of Cambridge, England, 1989.

[Wilkins, 1988] D.E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers, San Francisco, CA, 1988.

[Williams and Baird, 1993a] R. J. Williams and III Baird, L. C. Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning systems. Technical report, Boston: Northeastern University, College of Computer Science, 1993. Technical Report NU-CCS-93-11; available at http://www.ccs.neu.edu/home/rjw/pubs.html.

[Williams and Baird, 1993b] R. J. Williams and III Baird, L. C. Tight performance bounds on greedy policies based on imperfect value functions. Technical report, Boston: Northeastern University, College of Computer Science, 1993. Technical Report NU-CCS-93-14; available at http://www.ccs.neu.edu/home/rjw/pubs.html.