Handling Uncertainty Systems in the Situation Calculus with Macro-actions

by

Yilan Gu

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

Abstract

Handling Uncertainty Systems in the Situation Calculus with Macro-actions

Yilan Gu

Master of Science

Graduate Department of Computer Science

University of Toronto

2002

The situation calculus is a second-order logic language used to describe the characteristics of autonomous agents acting in a dynamic system. Its breadth and powerfulness has been shown by the tremendous work achieved in broad areas. Recently, researchers have become more and more interested in modeling and controlling the performance of agents in an uncertainty system with such language.

In this paper, we focus on the problems that the autonomous agent performs similar strategies repeatedly under same local situations in some uncertainty system. We introduce a special concept of action — macro-action — into the situation calculus, extend basic action theories and regression operators, and develop a knowledge base for the macro-actions so that the agent can remember certain information of them and later "recall" it when the agent performs the macro-actions in the same local situation again, therefore achieving the goal of saving computational time.

Acknowledgements

I am very grateful to Raymond Reiter, my supervisor, for his patient guidance and helpful advice at every stage of my work. His professional supervision makes me feel comfortable and enjoyable during the period of studies. Besides, I also want to thank him for reading my thesis carefully even in a bad health condition. My thanks also go to Craig Boutilier, my vice-supervisor and second reader, for very useful discussion and reading the whole thesis patiently. I want to thank Hector Levesque for taking over the responsibility when Raymond is not available.

I would like to acknowledge Iluju Kiringa, YongMei Liu and Mikhail E. Soutchanski who gave me nice suggestions. Thanks also go to many other friends I met both in Canada and in China.

At last, but certainly not least, I would like to thank my parents and my brother for their unconditional love and support I could feel even from the distance.

Contents

Chapter 1

Introduction

Since the birth of artificial intelligence (AI) in the mid-1950s [20, 5], developing techniques for constructing robust, autonomous agents that are able to achieve good performance in complex, real-world environments is always a central goal of AI. In order to achieve such a goal, researchers study simulated actions of autonomous agents in dynamic environments and try to develop formalized high-level controllers for autonomous agents. Several logic-based action formalisms have been developed to facilitate describing dynamic systems, such as the situation calculus [18, 17, 28], features and fluents [29], $\mathcal{A}$ calculus [11] and event calculus [14, 31]. The situation calculus is one of the oldest and most powerful languages. After the solving of frame problem in the situation calculus [27], the research on the applications of situation calculus and further implementations now become a very active area in AI. The *Cognitive Robotics* group [1] of the University of Toronto has been working on it and proposed a programming language called *Golog* [15] which "appears to offer significant advantage over current tools for applications in dynamic domains like the high-level programming of robots and software agents, process control, discrete event simulation, complex database transactions, etc" [28].

To make autonomous agents perform "intelligently" in the real world, especially in an uncertainty environment, we not only study the relationship between actions and dynamic world, but also need to get known about how human beings act intelligently in a dynamic world, so that the autonomous agents can simulate human behaviors and act "intelligently" (although AI is not always about simulating human intelligence [19]). As we know, in the real world, people often meet with unpredictable situations. For instance, when we flip a coin, we can not foretell the outcome is head or tail. So is it for autonomous agent, it is natural to require an autonomous agent to fit in such uncertainty system. Hence, from the end of 90's of last century, based on the study of the situation calculus and logic with probability [10], researchers began to work on high-level programming for robot acting in probabilistic uncertainty systems with stochastic actions. From different points of view, they give several kinds of study results. For example, an extended interpreter of Golog called *stGolog* (c.f. [28] Chapter 12) deals with derived probabilities and expected values problems for robots with probabilistic uncertain outcomes of actions. The *dtGolog* [4] is another extended interpreter of Golog dealing with decision-making problems. C. Boutilier, R. Reiter and B. Price proposed symbolic dynamic programming for first-order Markov Dedision Processes (FOMDPs) [3], which is a new approach using the situation calculus to deal with problems modeled in FOMDPs. All these studies not only show the possibility of dealing with uncertainty dynamic world by using the logic programming language, but also show the contribution of planning and decision-making theory to high-level robotic control.

The research of uncertainty systems in the situation calculus till now is based on primitive actions. Every step of regression and computation of probabilities needs to be repeated even if we compute for same sequences of actions under similar situations at different time. However, we notice that a robot, or more generally, an autonomous agent often works under a similar environment and is asked to solve similar problems, which involves lots of repetition in actions and outcomes of probabilities. For example, if we ask an autonomous agent to climb a hundred continuous same-height stairs, it can be viewed as that the agent repeats a certain sequence of actions at the same situation hundreds of times provided that we can reset the agent's situation to be the initial situation if malfunctions occur. Inspired by the idea of reuse of local policies for local Markov decision processes (MDPs) [25, 13, 21, 32], we here are trying to consider certain combinations of primitive actions described in the situation calculus as a whole , preprocess its properties, and later reuse the outcomes as if the agent has "learned" the knowledge, therefore make the agent become "cleverer".

We will begin with some background review of the situation calculus, the Golog and the stGolog in Chapter 2. And then, in Chapter 3, by giving an example of robot climbing stairs, we lead the discussion to treating certain complex actions as a whole, naming them as *macro-actions*, and discuss the change of basic action theories on them as well as the extended probabilities for uncertainty systems. In Chapter 4, we develop a knowledge base (static part) for macro-actions by using extended regression operator. After developing the static part of the knowledge base, we give an interpreter modified from the stGolog for programs that might include macro-actions in Chapter 5, and discuss the benefits and limitations of using macro-actions based on experiments. We end up with conclusions and future work in Chapter 6.

Chapter 2

Literature Review and Background

Since much of the proposed work is predicated on the high-level agent control in the uncertainty system, a review of the technical and historical background for the work of remaining chapters is presented here.

At least two aspects need to be addressed while modeling the behavior of an agent acting in an uncertainty system. First, the situation calculus and the basic action theory give us the power to model the dynamic world, the actions of the agent and their effects. Second, concerning the uncertainty system, we need a good understanding of how probabilistic uncertainty is expressed in the situation calculus.

2.1 The Language of the Situation Calculus

The basic conceptual and formal ingredients of the situation calculus were first proposed by John McCarthy in 1963 [18]. Based on several researchers' study and proposals [22, 6, 12, 30, 9], Ray Reiter [27] provided a solution to the frame problem observed by John McCarthy and Pat Hayes [18] and systematically described the situation calculus-based approaches to modeling dynamic world [28]. In the last ten years, under the leadership of Ray Reiter and Hector Levesque, the Cognitive Robotics Group [1] at

University of Toronto uses the situation calculus as a foundation for practical work in planning, control, simulation, etc, which disabuses some limiting view of the situation calculus [28]. Their work draws researchers' attention, and the situation calculus becomes more popular in the AI area.

The language of the situation calculus $\mathcal{L}_{sc}$ that we adopt here is from [28], which is a second-order language specifically designed for representing dynamically changing world. It is a three-sorted language with equality. The three disjoint sorts of $\mathcal{L}_{sc}$ are:

- *action:* a first-order term representing actions in dynamic world, such as `jump` (the action of jumping), $\texttt{kick(}x\texttt{)}$ (kicking object x), and $\texttt{put(}r, x, y\texttt{)}$ (robot r putting object x on top of object y), etc. The constant and function symbols for actions are completely application-dependent.

- *situation:* a first-order term which denotes possible world histories. A distinguished constant S_0 and function symbol *do* are used. S_0 denotes the *initial situation*, before any action has been performed; $do(a, s)$ denotes the situation that results from performing action a in situation s.

- *object:* a catch-all sort representing for everything else depending on the domain of application, such as `ball, Mary`, etc.

In fact, every situation corresponds to a sequence of actions. For example, the initial situation S_0 corresponds to empty sequence of actions, the situation

$$do(\texttt{pickup}(x), do(\texttt{drop}(y), do(\texttt{pickup}(y), S_0)))$$

corresponds to the action sequence $\texttt{pickup(}y\texttt{),drop(}y\texttt{),pickup(}x\texttt{)}$ from the beginning. Moreover, we will use binary relation $s \sqsubset s'$ to represent that s is a proper sub-history of s', and $s \sqsubseteq s'$ is equivalent to $s = s' \vee s \sqsubset s'$.

Another important term in the situation calculus is *fluent*. *Fluents* are predicates and functions whose values may vary from situation to situation, used to describe what holds in a situation. By convention, the last argument of a fluent is a situation. For example, the fluent $\texttt{Holding}(r, x, s)$ might stand for the relation of robot r holding object x in situation s.

The logical symbols of the language are $\neg, \wedge, \exists$. Other connectives and the universal quantifier are the usual abbreviations.

Finally, a distinguished predicate $Poss(A, s)$ is used to state that the action A can be performed in situation s. For example, $Poss(\texttt{pickup}(r, x), S_0)$ says that the robot is able to pick up object x in the initial situation.

This completes the specification of the language $\mathcal{L}_{sc}$. For later convenience, an abbreviation is introduced as follows:

Abbreviation 2.1 ([28] Chapter 4.5)

- $do([], s) \stackrel{def}{=} s$;
- $do([a_1, a_2, \cdots, a_n], s) \stackrel{def}{=} do(a_n, do(\cdots, do(a_1, s) \cdots))$.

And $[a_1, a_2, \cdots, a_n]$ *is called a* log.

Notice that there is a one-to-one correspondence between a log beginning at the initial time and a situation, whenever this log is finite or infinite.

2.2 The Basic Action Theory

A *basic action theory* is a set of axioms represented in the situation calculus to model the actions and their effects in a given dynamic system $\mathcal{D}$ together with *functional fluent consistency property*. Hereby we just present a summary, the detailed explanation could be found in [24, 28].

The set $\mathcal{D} = \mathcal{D}_f \bigcup \mathcal{D}_{ap} \bigcup \mathcal{D}_{ss} \bigcup \mathcal{D}_{una} \bigcup \mathcal{D}_{S_0}$ consists of following axioms:

- Fundamental axioms, denoted as $\mathcal{D}_f$.

  There are four axioms included in $\mathcal{D}_f$ [24]. For instance, $\neg s \sqsubset S_0$ represents no situation is a proper history of the initial situation.

Since the fundamental axioms is so mechanical, we will not write them out during description, but assume them to be true for any basic action theory.

- Action precondition axioms, denoted as $\mathcal{D}_{ap}$.

  For each action function (could be 0-ary) A, there is one axiom of the form:

$$Poss(a(\vec{x}), s) \equiv \Pi_a(\vec{x}, s),$$

  where $\Pi_a(\vec{x}, s)$ is a formula *uniform in s* (cf. Appendix B), $\vec{x} = x_1, x_2, \cdots, x_n$ for some natural number n (if $n = 0$, $a(\vec{x})$ is 0-ary A) and all atomic propositions in it are fluents. This axiom characterizes the preconditions of performing action A in the current situation s.

- Successor state axioms, denoted as $\mathcal{D}_{ss}$.

  A successor state axiom for an $(n+1)$-ary ($n \in \{0, 1, 2, \cdots\}$) relational fluent F is a sentence of $\mathcal{L}_{sc}$ of the form:

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

  where $\Phi_F(\vec{x}, a, s)$ is a formula uniform in s, $\vec{x} = x_1, x_2, \cdots, x_n$ (if $n = 0$, F has one parameter of sort situation) and all atomic propositions in it are ground fluents or of form $a = a_i$ where a_i is some action.

  Similarly, a successor state axiom for an $(n+1)$-ary functional fluent f is a sentence of $\mathcal{L}_{sc}$ of the 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 is s.

  The successor state axiom for fluent F(respectively f) completely characterizes the value of fluent F in the successor resulting from performing primitive action a in situation s.

- Unique name axioms, denoted as $\mathcal{D}_{una}$.

  For any n-ary action a, $a(x_1, \cdots, x_n) = a(y_1, \cdots, y_n) \supset x_1 = y_1 \wedge \cdots \wedge x_n = y_n$; for any distinct actions a and b, $a(\vec{x}) \neq b(\vec{y})$.

  Since the unique name axioms is so mechanical, we will not write them out during description, but assume them to be true for any basic action theory.

- Initial database, denoted as $\mathcal{D}_{S_0}$.

  It is a set of first order sentences in which S_0 is the only term of the situation sort (i.e., uniform in S_0). No sentence of $\mathcal{D}_{S_0}$ quantifies over situations, or mentions $Poss$ or the function symbol do. Notice that the initial database may contain sentences mentioning no situation term at all, for example, unique names for individuals, or "timeless" facts like $dog(x) \supset mammal(x)$.

Finally, the *functional fluent consistency property* is as follows:

Suppose f is a functional fluent whose successor state axiom in $\mathcal{D}_{ss}$ is

$$f(\vec{x}, do(a, s)) = y \equiv \phi_f(\vec{x}, y, a, s),$$

then

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

The reason of requiring this consistency property is that it provides a sufficient condition for preventing a source of inconsistency in f's successor state axiom.

Notice that the models we consider in this report all satisfy the *Markov* property – the truth values of the fluents at next situation are dependent only on the action and the truth values of the fluents at current situation.

2.3 Complex actions, Procedures and Golog

So far, in our treatment of the situation calculus, we only talked about primitive actions, with effects and preconditions independent of each other. In this section we will introduce a kind of compositional treatment of the frame problem for complex actions, i.e., actions that have other actions as components. This results in a novel kind of high-level programming language – Golog [16].

To handle complex actions, it is sufficient to show that for each complex action δ we care about, there is a ternary relation in the situation calculus, which we call $Do(\delta, s, s')$, is an abbreviation for a situation calculus formula which indicates that complex action δ, when started in situation s, can terminate legally in situation s'. Here δ is of one of the following actions:

1. *Primitive action: a (a might have parameters).*
2. *Sequence:* $\alpha;\beta$. Do action α, followed by action β.
3. *Test action:* $p?$. Test the truth value of expression p in the current situation.
4. *Nondeterministic action choice:* $\alpha|\beta$. Do α or do β.
5. *Nondeterministic choice of arguments:* $(\pi x)\alpha(x)$. Nondeterministically pick a value for x, and for that value of x, do action $\alpha(x)$.
6. *Conditionals: if-then-else* and *while* loops.
7. *Procedures, including recursion.*

Because of the definition of complex actions, we then can deal with nondeterministic, conditional, or concurrent operations. Detailed explanation and examples can be found in [28]. Golog program is a procedure defined as following:

Definition 2.2 ([28] Chapter 6.1.1) *A* Golog program *is of form:*

`proc` $P_1(\vec{v_1})\delta_1$ `endproc`$;\cdots;$ `proc` $P_n(\vec{v_n})\delta_n$ `endproc`$;\ \delta_0$

where P_i is declaration of procedures with formal parameter $\vec{v_i}$ and procedure body δ_i for each $i(1 \leq i \leq n)$, δ_0 is the main program body. $\delta_0, \cdots, \delta_n$ are complex actions, *extended by actions for procedure calls, as described in above Definition.*

The semantics of a program is

$$Do(\{\texttt{proc}\ P_1(\vec{v_1})\delta_1\ \texttt{endproc};\cdots;\ \texttt{proc}\ P_n(\vec{v_n})\delta_n\ \texttt{endproc};\ \delta_0\}, s, s') \stackrel{def}{=}$$
$$(\forall P_1, \cdots, P_n).[\bigwedge_{i=1}^{n}(\forall s_1, s_2, \vec{v_i}).Do(\delta_i, s_1, s_2) \supset P_i(\vec{v_i}, s_1, s_2)] \supset Do(\delta_0, s, s'),$$

i.e. when $P_1, \cdots, P_n$ are the smallest binary relations on situations that are closed under the evaluation of their procedure bodies $\delta_1, \cdots, \delta_n$, then any transition (s, s') obtained by evaluating the main program δ_0 is a Golog transition for the evaluation for the program.

Golog appears to offer significant advantages over current tools for applications in dynamic domains like the high-level programming of robots and software agents, process control, discrete event simulation, complex database transactions, etc [2, 7].

2.4 The Regression Operator

Regression is a central computational mechanism that forms the basis for many planning procedures (Waldinger [33]) and for automated reasoning in the situation calculus (Pednault [23], Pirri and Reiter [24]). Roughly speaking, the regression of a formula ϕ through an action a is a formula ϕ' that holds prior to a being performed iff ϕ holds after a. Successor state axioms support regression in a natural way. In [28], Reiter introduces a notation $\mathcal{R}$ as regression operator, and defines the regression of a *regressable* formula W of $\mathcal{L}_{sc}$ as follows:

Definition 2.3 ([28] Definition 4.5.1) *A formula W of $\mathcal{L}_{sc}$ is* `regressable` *iff*

1. *Every term of sort situation mentioned by W has the syntactic form $do([\alpha_1, \cdots, \alpha_n], S_0)$ for some $n \geq 0$, and for terms $\alpha_1, \cdots, \alpha_n$ of sort action.*

2. *For every atom of the form $Poss(\alpha, \sigma)$ mentioned by W, α has the syntactic form $A(t_1, \cdots, t_n)$ for some n-ary function symbol A of $\mathcal{L}_{sc}$.*

3. *W does not quantify over situations.*

4. *W does not mention the predicate symbol $\sqsubset$, nor does it mention any equation atom $\sigma = \sigma'$ for terms σ, σ' of sort situation.*

Definition 2.4 ([28] Definition 4.7.2)

1. *Suppose $W = Poss(a(\vec{t}), \sigma)$ where $a(\vec{t})$ and σ are of sort action and situation respectively, and we have action precondition axiom of form*

$$Poss(a(\vec{x}), s) \equiv \Pi_a(\vec{x}, s) \ ,$$

without loss of generality, assume that all quantifiers (if any) of $\Pi_a(\vec{x}, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $Poss(a(\vec{t}), \sigma)$, then

$$\mathcal{R}[W] = \mathcal{R}[\Pi_a(\vec{t}, \sigma)] \ .$$

2. *Suppose W is a regressable atom, but not a Poss atom. There are three possibilities:*

(a) *S_0 is the only term of sort situation (if any) mentioned by W, then*

$$\mathcal{R}[W] = W \ .$$

(b) *Suppose that W mentions a term of the form $g(\vec{t}, do(\alpha', \sigma'))$ for some functional fluent g, and α' and σ' are of sort action and situation respectively. $g(\vec{t}, do(\alpha', \sigma'))$ mentions a* prime functional fluent *[28] term of form $f(\vec{r}, do(\alpha, \sigma))$ where α and σ are of sort action and situation uniform in S_0 respectively. Suppose f's successor state axiom in $\mathcal{D}_{ss}$ is*

$$f(\vec{x}, do(a, s)) = y \equiv \phi_f(\vec{x}, y, a, s) \ .$$

Without loss of generality, assume that all quantifiers (if any) of $\phi_f(\vec{x}, y, a, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $f(\vec{r}, do(\alpha, \sigma))$. Then

$$\mathcal{R}[W] = \mathcal{R}[(\exists y).\phi_f(\vec{r}, y, \alpha, \sigma) \wedge W|_y^{f(\vec{r}, do(\alpha, \sigma))}] \ .$$

Here y is a variable not occurring free in $W, \vec{r}, \alpha$ or σ.

(c) *W is a relational fluent atom of form $F(\vec{t}, do(\alpha, \sigma))$ where α and σ are of sort action and situation respectively. Let F's successor state axiom in $\mathcal{D}_{ss}$ be*

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

Without loss of generality, assume that all quantifiers (if any) of $\Phi_F(\vec{x}, a, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $F(\vec{t}, do(\alpha, \sigma))$. Then

$$\mathcal{R}[W] = \mathcal{R}[\Phi_F(\vec{t}, \alpha, \sigma)] .$$

3. *For non-atomic formulas, regression is defined inductively as follows.*

$$\mathcal{R}[\neg W] = \neg \mathcal{R}[W]$$

$$\mathcal{R}[W_1 \wedge W_2] = \mathcal{R}[W_1] \wedge \mathcal{R}[W_2]$$

$$\mathcal{R}[(\exists x)W] = (\exists x)\mathcal{R}[W]$$

2.5 Stochastic Actions, Probability and stGolog

In this work, we will concentrate on the dynamical systems with uncertainty. In an uncertainty system, *stochastic actions*, actions with with uncertain outcomes that an agent can perform, are introduced. For example ([28]) in text, stochastic action $go(l)$ means that the robot goes to location l, and the performance of $go(l)$ ends up with two outcomes: one is $endUpAt(l)$ meaning that the robots ends up at location l, the other is $getLost(l)$ meaning that the robot gets lost in the process of going to location l. These outcomes are nature's choices, i.e., not under the control of the robot. Notationally, we characterize this setting by:

$$choice(go(l), a) \stackrel{def}{=} a = endUpAt(l) \vee a = getLost(l).$$

All the nature's choices of stochastic actions are primitive actions and the action precondition axioms and successor state axioms are presented for every primitive action. Moreover, we need to represent the probability of each outcome of a stochastic action. We must require that whenever one of nature's action's preconditions is false, the action will have zero probability, i.e.,

$$\begin{aligned} prob(a,\beta,s)=p \;\stackrel{def}{=}\; & choice(\beta,a)\wedge Poss(a,s)\wedge p=prob_0(a,\beta,s)\vee \\ & [\neg choice(\beta,a)\vee\neg Poss(a,s)]\wedge p=0. \end{aligned} \tag{2.1}$$

Here, $prob_0(a,\beta,s)$ is a specification of the probability that a is selected in the situation s and the outcome of stochastic action β, given that a is one of the nature's choices for β and, moreover, that *a is possible in s* [28].

It is axiomatizer's responsibility to ensure that a proper probability distribution has been defined while formalizing a probabilistic domain in the situation calculus. One needs to verify the following two properties:
for any stochastic action α and its nature's choices A_i $(i=1,2,\ldots,k)$,

$$(a)\quad Poss(A_i,s)\supset prob_0(A_i,\alpha,s)>0,\;\; i=1,2,\ldots,k. \tag{2.2}$$

$$(b)\quad Poss(A_1,s)\vee\cdots\vee Poss(A_k,s)\supset\sum_{i=1}^{k} prob(A_i,\alpha,s)=1. \tag{2.3}$$

Based on above extensions, new programs, named *stGolog program* [28] are constructed from stochastic actions together with the Golog program constructors sequence, tests, while loops, conditionals and procedures. stGolog program do *not* involve any form of nondeterminism; neither the nondeterministic choice, |, of two actions, nor the π operator are allowed. Moreover, notice that there is a dummy symbol *nil* is introduced into sequence indicating the end of the sequence, which is another difference from Golog sequence. An stGolog interpreter (Appendix A) [28] is developed via $stDo(\alpha,p,s,s')$

meaning that agent performs stGolog program (or actions) α at the situation s, by nature's choices, it may ends at situation s' with probability p. With the help of $stDo$, the probability that some situation-suppressed sentence ψ will be true after executing stGolog program γ:

$$probF(\psi,\gamma) \stackrel{def}{=} \sum_{\{(p,\sigma)\ \mid\ \mathcal{D}\models stDo(\gamma:nil,p,S_0,\sigma)\wedge\psi(\sigma)\}} p, \tag{2.4}$$

where $\mathcal{D}$ stands for the background basic action theory.

We also can introduce cost and reward of actions, exogenous events and uncertain initial situation into stGolog [28]. To simplify problems we will meet with, we ignore them here. Up till now, we finished reviewing almost all the knowledge background that our later discussion will based on.

At last, for later convenience, we use the following conventions:

1. In an uncertainty system modeled in the situation calculus, we use upper-case letters (with or without subscript and superscript) to denote the deterministic actions, use lower-case letters (with or without subscript and superscript) to denote variables of the deterministic actions, and use α, β (with or without subscript and superscript) to denote any kinds of actions including either stochastic or deterministic, primitive or complex, instance or variable. When we say α is of sort action, we mean that α is either instance or variable of a deterministic primitive action.

2. For the sake of the convenience for expressions and notations, except specific announcement (e.g. in an example), for an uncertainty system modeled in the situation calculus, we may omit the free variables appearing in the action functions. That is to say, n-ary action function, say $a(\vec{x})$, for some natural number n (whether deterministic or stochastic, primitive or complex) will be often denoted as a later.

Chapter 3

Introducing Macro-actions into the Uncertainty System

As we saw in the previous chapter, the basic action theories as well as the theory of probability provide a convenient way for us to deal with high-level robot control in uncertainty systems. However, we want to make the autonomous agent become "cleverer" in the uncertainty world in the sense that it can remember what it did before under same environment. Therefore, similar to an intelligent human being, the agent won't waste time on re-computation. In this chapter, we will discuss the motivation of the work in this paper explicitly and then start the first step of reaching the object of making robot "cleverer".

3.1 Example of Climbing Stairs and the Motivation

One of the main purposes of creating intelligent autonomous agents is to make them serve and help human beings efficiently on particular topics such as exploring volcanos, assisting disabled people at home, and making products in the factories. These agents although "living" in uncertainty systems, still meet lots of similar situations, work on the same tasks and repeat the same strategies most of the time. Let us first look at a simple example as follows:

Example 3.1 Consider a robot with two legs, *main* and *supporting*, is asked to climb stairs. We first declare following hypotheses:

(1) The main leg has thigh, shin and foot, and we will describe their actions in detail.

(2) We ignore most actions of supporting leg's thigh and shin, and will simplify the actions to only one action.

(3) The stairs are much lower than the legs' knees.

(4) The width of every stair is short enough so that the robot is always directly in front of the new stair after a previous sequence of climbing actions.

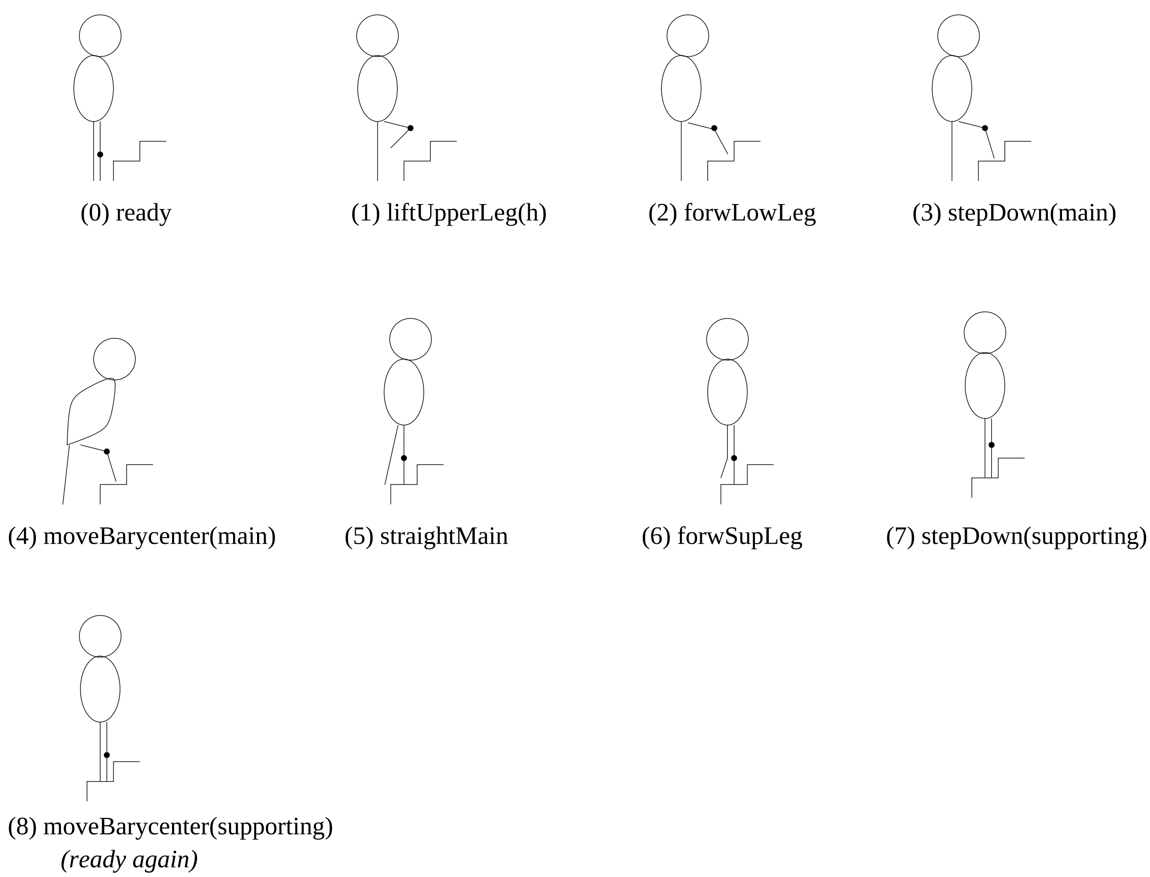

Figure 3.1: The Decomposion of Actions of Robot Climbing Stairs

In detail, the following stochastic actions are concerned when we describe a robot climbing stairs (cf. Figure 3.1):

- $liftUpperLeg(h)$, meaning that the robot lifts the thigh of the main leg, is decomposed into the following two nature's choices:
  - $liftTill(h)$: the thigh is lifted successfully till the knee's height is h to current stair;
  - $malfunc(h)$: malfunction occurs when robot lifts its thigh till the knee's height is h to current stair, then the main leg's position will end up at $wrongPos$, meaning wrong position, which represents an abnormal position unsuitable for performing further actions.
- $forwLowLeg$, meaning that the robot moves forward the shin of the main leg, is decomposed into the following two nature's choices:
  - $forwLowLegS$: the action $forwLowLeg$ performs successfully;
  - $forwLowLegF$: malfunction occurs and the action $forwLowLeg$ fails.
- $stepDown(l)$, meaning that the robot steps the leg and foot of leg l down straightly till it touches the surface of the stair, is decomposed into the following two nature's choices:
  - $stepDownS(l)$: the action $stepDown(l)$ performs successfully;
  - $stepDownF(l)$: malfunction occurs and the action $stepDown(l)$ fails.
- $moveBarycenter(l)$, meaning that the robot moves its barycenter onto leg l, is decomposed into the following two nature's choices:
  - $moveBarycenterS(l)$: the action $moveBarycenter(l)$ performs successfully;
  - $moveBarycenterF(l)$: malfunction occurs and the action fails.
- $straightLeg$ is actually a deterministic action, meaning that the robot straightens his main leg (its side effect is that the supporting leg leaves the ground).
- $forwSupLeg$, meaning that the robot moves forward his supporting leg, is decomposed into the following two nature's choices:
  - $forwSupLegS$: the action $forwSupLeg$ performs successfully;
  - $forwSupLegF$: malfunction occurs and the action $forwSupLeg$ fails.

Notice that the *current stair to leg* l is defined as whose ground the foot of leg l is on, or whose ground the foot of leg l was on before this foot touches a surface of some other stair again; and the *new stair to leg* l is defined as the next stair in front of current one, i.e., it will always be "new" to leg l before the foot of leg l touches the ground of the *new stair*. Now, we introduce following fluents to specify the environment:

- Relational fluent $straightMain(s)$: the main leg is straight.
- Relational fluent $barycenter(l, s)$: the barycenter of the robot is on leg l.
- Relational fluent $footOnGround(l, s)$: the foot of leg l is on the ground.
- Relational fluent $overNewStair(l, s)$: the leg l is over the new stair to leg l.
- Functional fluent $mainToCurr(s)$: either is the height of the main leg's the foot to the ground of the current stair to $main$, or becomes a special term $wrongPos$ if stochastic action $liftUpperLeg(H)$ for some H doesn't perform successfully; and once the robot's $main$ leg is in $wrongPos$, it stays.

Because nature's actions above are all deterministic, it is predictable how they change the state of the world.

$$\begin{aligned} straightMain(do(a,s)) \ &\equiv \ a = straightLeg \vee \\ & \quad straightMain(s) \wedge \neg(\exists h) a = liftTill(h), \end{aligned}$$

$$\begin{aligned} barycenter(l, do(a,s)) \ &\equiv \ a = moveBarycenterS(l) \vee barycenter(l,s) \wedge \\ & \quad \neg(\exists l')[a = moveBarycenterS(l') \wedge l \neq l'], \end{aligned}$$

$$\begin{aligned} & footOnGround(l, do(a,s)) \ \equiv \\ & \qquad a = stepDownS(l) \vee footOnGround(l,s) \wedge [l = main \wedge \\ & \qquad \neg(\exists h) a = liftTill(h) \vee l = supporting \wedge a \neq straightLeg], \end{aligned}$$

$$\begin{aligned} & overNewStair(l, do(a,s)) \ \equiv \\ & \qquad a = forwLowLegS \wedge l = main \vee a = forwSupLegS \wedge l = supporting \vee \\ & \qquad overNewStair(l,s) \wedge a \neq stepDownS(l), \end{aligned}$$

$$
\begin{aligned}
&mainToCurr(do(a,s)) = h \equiv \\
&\qquad a = stepDownS(main) \wedge h = 0 \vee (\exists h')a = malfunc(h') \wedge h = wrongPos \vee \\
&\qquad a = liftTill(h) \vee mainToCurr(s) = wrongPos \wedge h = wrongPos \vee \\
&\qquad mainToCurr(s) \neq wrongPos \wedge \neg(\exists h')a = malfunc(h') \wedge \\
&\qquad a \neq stepDownS(main) \wedge \neg(\exists h')[a = liftTill(h') \wedge h \neq h'] \wedge \\
&\qquad h = mainToCurr(s).
\end{aligned}
$$

Moreover, the action preconditions for nature's actions are specified as follows:

$$
\begin{aligned}
&Poss(liftTill(h), s) \equiv barycenter(supporting, s), \\
&Poss(malfunc(h), s) \equiv barycenter(supporting, s), \\
&Poss(forwLowLegS, s) \equiv \quad \neg mainToCurr(wrongPos, s) \wedge \\
&\qquad\qquad\qquad\qquad\qquad \neg footOnGround(main, s), \\
&Poss(forwLowLegF, s) \equiv \quad \neg mainToCurr(wrongPos, s) \wedge \\
&\qquad\qquad\qquad\qquad\qquad \neg footOnGround(main, s), \\
&Poss(stepDownS(l), s) \equiv \neg footOnGround(l, s) \wedge overNewStair(l, s), \\
&Poss(stepDownF(l), s) \equiv \neg footOnGround(l, s) \wedge overNewStair(l, s), \\
&Poss(moveBarycenterS(l), s) \equiv footOnGround(l, s), \\
&Poss(moveBarycenterF(l), s) \equiv footOnGround(l, s), \\
&Poss(straightLeg, s) \equiv \quad \neg straightMain(s) \wedge footOnGround(main, s) \\
&\qquad\qquad\qquad\qquad\quad \wedge barycenter(main, s), \\
&Poss(forwSupLegS, s) \equiv barycenter(main, s) \wedge straightMain(s), \\
&Poss(forwSupLegF, s) \equiv barycenter(main, s) \wedge straightMain(s).
\end{aligned}
$$

For example, the robot is possible to lift the thigh of his main leg iff its barycenter is on his supporting leg; it can attempt to move forward the shin of his main leg iff his main leg is not in a wrong position and the foot of his main leg is not on the ground; etc.

Moreover, since we are characterizing an uncertainty model, we need to declare the following probabilities:

$$prob_0(liftTill(h), liftUpperLeg(h), s) \stackrel{def}{=} 100/(h + 100),$$

$$prob_0(malfunc(h), liftUpperLeg(h), s) \stackrel{def}{=} h/(h + 100),$$

$$prob_0(forwLowLegS, forwLowLeg, s) \stackrel{def}{=} 80/(mainToCurr(s) + 80),$$

$$prob_0(forwLowLegF, forwLowLeg, s) \stackrel{def}{=} mainToCurr(s)/(mainToCurr(s)+80),$$

$$prob_0(stepDownS(l), stepDown(l), s) \stackrel{def}{=} 0.9,$$

$$prob_0(stepDownF(l), stepDown(l), s) \stackrel{def}{=} 0.1,$$

$$prob_0(moveBarycenterS(l), moveBarycenter(l), s) \stackrel{def}{=} 0.8,$$

$$prob_0(moveBarycenterF(l), moveBarycenter(l), s) \stackrel{def}{=} 0.2,$$

$$prob_0(straightLeg, straightLeg, s) \stackrel{def}{=} 1.0,$$

$$prob_0(forwSupLegS, forwSupLeg, s) \stackrel{def}{=} 0.8,$$

$$prob_0(forwSupLegF, forwSupLeg, s) \stackrel{def}{=} 0.2.$$

At last, we have following complete description of the initial database:

$$straightMain(S_0),$$

$$barycenter(supporting, S_0),$$

$$footOnGround(l, S_0) \equiv l = main \lor l = supporting,$$

$$\neg overNewStair(l, S_0),$$

$$mainToCurr(0, S_0),$$

$$legalStair(h) \equiv number(h) \land 0 < h < 20,$$

where predicate $number(h)$ means that h is a real number. Then, the following sequence of stochastic actions

$$\begin{array}{l} \texttt{proc}\ climbing(h) \\ \quad ?(legalStair(h)); liftUpperLeg(h); forwLowLeg; stepDown(main); \\ \quad moveBarycenter(main); straightLeg; forwSupLeg; \\ \quad stepDown(supporting); moveBarycenter(supporting) \\ \texttt{endproc} \end{array} \tag{3.1}$$

describes the actions that the robot need to execute when it climbs a stair, and it climbs the legal stair successfully iff the log

$$[liftTill(h), forwLowLegS, stepDownS(main), moveBarycenterS(main), straightLegS,$$
$$forwSupLegS, stepDownS(supporting), moveBarycenterS(supporting)]$$

is performed by nature's choices when the above sequence of stochastic actions is requested to be performed by the agent.

Thinking of human beings, when they climb stairs, they don't care how many stairs they've climbed or how they have climbed. As long as they know how to climb stairs and there is some stair in front of them, they will naturally repeat the sequence of climbing actions without "thinking". If they fall by accident (not very seriously injured of course), they will stand up again and repeat the sequence of actions. Similar for the autonomous agent here, we would like the robot to concentrate on the local status of climbing a stair, and provide that the controller can reset the robot to the initial status when malfunctions occur. If we ask the robot to climb stairs of same height n times (including the times that the robot is reset and requested to re-climb) without remembering how it climbs former stairs, we need to compute the probabilities of the nature's choices of above sequence repeatedly, and do regressions step by step again and again by using stGolog, since the methodology we have met before in the situation calculus is memoryless except the history of the log from the initial situations. We hope that the agent can have some "memory", and will "recall" the information it remembers, and therefore will perform climbing actions without "thinking" in some sense. To achieve this, our intuitive idea is considering certain types of complex actions in the situation calculus as a whole, performing some preprocessing (including extending the basic axioms and probabilities and save them as rules) and later reusing the saved informations for application. This is exactly what we are going to do in the later sections and chapters.

3.2 The Macro-actions

In this section, we focus on the uncertainty system with stochastic actions described above in Section 2.5. We are going to observe different types of complex actions in the situation calculus generally to see which kinds of complex actions can be consider as a whole (later, without ambiguity, called *macro-action* in the situation calculus), and get unique extended precondition axioms, extended successor state axioms, axioms of the probabilities for them, therefore for the purpose of reuse. In practical life, intelligent agents are often designed to deal with particular types of problems as the examples mentioned in the former section, meet with similar environments and perform similar sequence of actions repeatedly. Under such situations, to reuse the outcome of certain type of macro-actions for solving problems may bring us computational advantage.

3.2.1 Finding the Proper Structure of the Macro-actions

For the clarity, we restate here that we are dealing with the uncertainty system with stochastic actions $\alpha_1, \alpha_2, \cdots, \alpha_t$. And, we have the nature's choices $A_{i,1}, A_{i,2}, \cdots, A_{i,k_i}$ for stochastic action α_i (when $k_i = 1$, α_i is actually a deterministic action), the precondition axioms for every primitive deterministic action, successor state axioms for fluents involving primitive deterministic actions and probability of each nature's choice of stochastic actions as $prob_0$ given above in section 2.5.

Firstly, we will not consider disjunction "$|$" or existential quantification "π" as a part of macro-actions, because we are interested in probabilistic uncertainty and wish to obtain the explicit probabilistic information that we desire to keep for macro-actions. The logical uncertainty expressed with disjunction and existential quantification doesn't has such exact information. For example (cf. [28] chapter 12), after dropping a coin, there is *some* place on the floor where it will end up with, but we don't know where and how exact that place will be. Therefore, it is impossible for us to keep any numerical information for logical uncertainty.

Secondly, let's look at a complex action that we are obviously not easy to get a unique successor axiom for – the "*while*" loop. We may have a look at the following simple example.

Example 3.2 Suppose we are given stochastic action $flipcoin$ with the following basic axioms:

$$choice(a, flipcoin) \equiv a = fliphead \vee a = fliptail,$$

$$Poss(fliphead, s) \equiv true,$$

$$Poss(fliptail, s) \equiv true,$$

$$head(do(a, s)) \equiv a = fliphead \vee head(s) \wedge a \neq fliptail,$$

$$tail(do(a, s)) \equiv a = fliptail \vee tail(s) \wedge a \neq fliphead,$$

$$prob_0(fliphead, flipcoin, s) = 0.5,$$

$$prob_0(fliptail, flipcoin, s) = 0.5,$$

$$\neg head(S_0),$$

$$\neg tail(S_0).$$

And now, we may have following procedure:

`proc` $showHead \;\; while \neg head \; do \; flipcoin$ `endproc`

It is easy to see that there is no way for us to tell in advance how many times we need to flip the coin to satisfy the goal "showing head" under general situations. Although for any finite number of iterations we can foretell the exact probability, there could have infinite choices to achieve the conditions of the "while" loop. However, there is not enough space for us to keep the extended axioms and probabilities for all the possible finite iterations. Therefore, we will not consider the "while" loop as a part of macro-actions.

Thirdly, the sequence action operator ";" seems considerable for constructing the macro-actions we need. Intuitively, a finite sequence of stochastic actions is totally deterministic in some sense, therefore easy for us to trace its characters and effects.

Consider a finite sequence of stochastic actions $\alpha = \alpha_1; \alpha_2; \cdots; \alpha_n$ $(n \geq 1)$. We present following extended definition of *regressable* formulas and *prime* functional fluents.

Definition 3.3 *Suppose s is either the initial situation S_0 or a variable of sort situation. A formula W of $\mathcal{L}_{sc}$ is called* s-regressable for s *iff*

1. *every term of sort situation mentioned by W has the form $do([\alpha_1, \cdots, \alpha_n], s)$ for some $n \geq 0$ (special case: if $n = 0$, $do([\alpha_1, \cdots, \alpha_n], s) = s$) and for terms $\alpha_1, \cdots, \alpha_n$ of sort action;*

2. *other conditions are same as the $2^{nd}, 3^{rd}$ and 4^{th} conditions in the Definition 2.3.*

Definition 3.4 *Suppose s is either the initial situation S_0 or a variable of sort situation. A functional fluent term is* s-prime for s *iff it has the form $f(\vec{t}, do([\alpha_1, \cdots, \alpha_n], s))$ for $n \geq 1$ and each of the terms $\vec{t}, \alpha_1, \cdots, \alpha_n$ is uniform in s.*

Notice that the *regressable* formula defined in [28] is same as S_0-*regressable* formula and *prime* functional fluent [28] is same as S_0-*prime* functional fluent we defined here. Moreover, the concept *uniform* is defined in [28] as Definition 4.4.1 (also, cf. Appendix B). Similar to *Remark 4.7.1* in [28], we have

Remark 3.5 *Suppose that $g(\vec{\tau}, do(\alpha, \sigma))$ has the property that every term of sort situation that it mentions has the form $do([\alpha_1, \cdots, \alpha_n], s)$ for some $n \geq 0$. Then $g(\vec{\tau}, do(\alpha, \sigma))$ mentions a s-prime functional fluent term.*

We then can extend the regression operator $\mathcal{R}$ onto s-regressable formula W for some situation s as follows:

Definition 3.6

1. *Suppose $W = Poss(a(\vec{t}), \sigma)$ where $a(\vec{t})$ and σ are of sort action and situation respectively, $\mathcal{R}[W]$ is defined same as Definition 2.4.*

2. *Suppose W is a s-regressable atom, but not a Poss atom.*

   (a) *s is the only term of sort situation (if any) mentioned by W. Then*

   $$\mathcal{R}[W] = W \ .$$

   (b) *Otherwise, the definition of $\mathcal{R}[W]$ is same as Definition 2.4, except for changing "prime functional fluent" to be "s-prime functional fluent".*

3. *For non-atomic formulas, regression is defined same as Definition 2.4.*

The reason we still keep using notation $\mathcal{R}$ is because the definition above is same as Definition 2.4 when W is regressable, i.e., Definition 3.6 is only an extension of the original regression operator.

Now we return to discuss if we can extend the action precondition axioms, the successor state axioms and the probability axioms for complex action composed by using operator ";". Suppose we have a sequential action $A = A_1; A_2; \cdots ; A_n$, where A_i is primitive deterministic action for every $i \in \{1, 2, \cdots , n\}$, notice that

$$\begin{aligned} Do(A, s, s') &\equiv (\exists! s_0, s_1, \cdots , s_n).s_0 = s \wedge s_n = s' \wedge (\wedge_{i=1}^{n} s_i = do(A_i, s_{i-1})) \\ &\equiv s' = do([A_1, A_2, \cdots , A_n], s), \end{aligned}$$

hence, to distinguish from abbreviation *log* and later be convenient to establish extended axioms, we can extend the notation $do(A, s)$ where A is of sort primitive action to be $do(A_1; A_2; \cdots ; A_n, s)$ $(n \geq 1)$ for primitive actions $A_1, A_2, \cdots , A_n$, indicating the situation after taking deterministic sequential action $A_1; A_2; \cdots ; A_n$ in the situation s.

- The precondition axiom $Poss(A, s)$ can be extended as follows:

  $n = 1$ $Poss(A, s)$ is the precondition axiom given in $\mathcal{D}$ for action A;

  $n > 1$ $Poss(A, s)$, meaning that A can be performed in the situation s,

$$\begin{aligned} &= Poss(A_1; A_2; \cdots; A_n, s) \\ &\stackrel{def}{=} (\wedge_{i=2}^n Poss(A_i, do([A_1, \cdots, A_{i-1}], s))) \wedge Poss(A_1, s) \qquad (3.2) \\ &\equiv \mathcal{R}[(\wedge_{i=2}^n Poss(A_i, do([A_1, \cdots, A_{i-1}], s))) \wedge Poss(A_1, s)], \\ &\equiv \Pi(\vec{t}, s), \end{aligned}$$

  where $\Pi(\vec{t}, s)$ is a formula uniform in s obtained by regression and simplification. The brief proof of the property that the regression result of a s-regressable formula is uniform in s can be found in Appendix B.

- Suppose given sequential action variable $a = a_1; a_2; \cdots; a_n$ $(n \in \mathcal{N})$, where every a_i is a variable for primitive deterministic action for every $i \in \{1, 2, \cdots, n\}$, the successor state axiom of every relational fluent $F(\vec{x}, do(a, s))$ and of every functional fluent $f(\vec{x}, do(a, s))$ can be extended as follows:

  $n = 1$ $F(\vec{x}, do(a, s)) \equiv \phi_F(\vec{x}, a, s)$ is the given successor state axiom for relational fluent F; and $f(\vec{x}, do(a, s)) = y \equiv \phi_f(\vec{x}, y, a, s)$ is the given successor state axiom for functional fluent f;

  $n > 1$ $F(\vec{x}, do(a, s))$

$$\begin{aligned} &= F(\vec{x}, do(a_1; a_2; \cdots; a_n, s)) \\ &\equiv \mathcal{R}[F(\vec{x}, do([a_1, a_2, \cdots, a_n], s))] \\ &\equiv \psi_F(\vec{x}, a_1, a_2, \cdots, a_n, s) \end{aligned}$$

  for some ψ_F uniform in s, if F is relational fluent; and

  $f(\vec{x}, do(a, s)) = y$

$$\begin{aligned} &\equiv f(\vec{x}, do(a_1; a_2; \cdots; a_n, s)) = y \\ &\equiv \mathcal{R}[f(\vec{x}, do([a_1, a_2, \cdots, a_n], s)) = y] \\ &\equiv \Psi_f(\vec{x}, y, a_1, a_2, \cdots, a_n, s) \end{aligned}$$

  for some Ψ_f uniform in s, if f is functional fluent.

- Now we also need to extend the probability function(2.1) *prob* in the stGolog for a deterministic sequential action $A = A_1; A_2; \cdots ; A_m$ of stochastic sequential action $\alpha = \alpha_1; \alpha_2; \cdots ; \alpha_n$, which is denoted as $probMac$.

  $$probMac(A, \alpha, s) = p \stackrel{def}{=}$$
  $$choiceMac(\alpha, A) \wedge Poss(A, s) \wedge p = prob_0(A_1, \alpha_1, s)*$$
  $$prob_0(A_2, \alpha_2, do(A_1, s)) * \cdots * prob_0(A_m, \alpha_m, do([A_1, \cdots , A_{m-1}], s))$$
  $$\vee(\neg choiceMac(\alpha, A) \vee \neg Poss(A, s)) \wedge p = 0,$$

  in which we define predication $choiceMac$ as follows:

  $$choiceMac(\alpha, a) \stackrel{def}{=}$$
  $$a \in \{A_1; A_2 \cdots ; A_m \mid m \in \mathcal{N} \wedge 1 \leq m \leq n \wedge (\wedge_{i=1}^{m} choice(\alpha_i, A_i)\},$$

  and say that deterministic sequential action A is a nature's choice of α if $choiceMac(\alpha, A)$ is true. In fact, $choiceMac$ is an extension of *choice*, and $probMac$ is an extension of *prob* in the stGolog.

As in [28], to specify an appropriate probabilistic domain for an uncertainty system, we need to verify that a proper probability distribution has been defined, i.e., the axiomatizer must ensure the two propositions (2.2) and (2.3) described Section 2.5 are satisfied. Therefore, according to the specification above, we can prove several properties for the definition of $probMac$.

Lemma 3.7 *Let $\alpha = \alpha_1; \alpha_2; \cdots ; \alpha_n$ ($n \in \mathcal{N}$ and $n \geq 1$) be stochastic sequential actions, and A be deterministic sequential actions satisfying that*

$$choiceMac(\alpha, A) \equiv true.$$

Suppose properties (a) and (b) above have been verified, then the following sentences follow from these properties, and the definition of $probMac$:

1. *All probabilities for deterministic sequential actions are bounded by 0 and 1:*

   $$(\forall a, \vec{x}, s).0 \leq probMac(a, \alpha, s) \leq 1.$$

2. *All non-outcomes of* α *have probability* 0*:*

   $$(\forall a, \vec{x}, s).\neg choiceMac(\alpha, a) \supset probMac(a, \alpha, s) = 0.$$

3. *Nature's choices are possible iff they have non-zero probability:*

   $$(\forall \vec{x}, s).Poss(A, s) \equiv probMac(A, \alpha, s) > 0.$$

Proof: Straightforward from the definition of *probMac* and properties (a) and (b). ■

Definition 3.8 *For any stochastic sequential action* $\alpha = \alpha_1; \alpha_2; \cdots; \alpha_n$ *and situation* s*, we define following set*

$$maxPoss(\alpha, s) \stackrel{def}{=} \{A = A_1; A_2; \cdots; A_m \mid choiceMac(\alpha, A) \wedge Poss(A, s) \wedge$$
$$(m = seqLength(\alpha) \vee m < seqLength(\alpha) \wedge ((\forall a).choice(\alpha_{m+1}, a) \supset \neg Poss(A; a, s)))\},$$

where the predicate $seqLength(a)$*, meaning the length of* a*, for macro-action or sequential action* a *is recursively defined as follows:*

1. *if* a *is a deterministic or stochastic action, then* $seqLength(a) = 1$*;*
2. *if* a *is of form* $\alpha; \beta$*, then* $seqLength(a) = seqLength(\alpha) + seqLength(\beta)$*.*

Intuitively, set $maxPoss(\alpha, s)$ is a collection of the maximal possible performable choices of α in the situation s. Then, we have following property.

Theorem 3.9 *In the probabilistic domain specified properly satisfying above two conditions (a) and (b), for any stochastic sequential action* $\alpha = \alpha_1; \alpha_2; \cdots; \alpha_n$*, we have*

$$\bigvee (A = A_1, \cdots, A_n \wedge choiceMac(\alpha, A) \wedge Poss(A, s)) \supset$$

$$\sum_{A \in maxPoss(\alpha, s)} probMac(A, \alpha, s) = 1.$$

Proof: We will prove it by complete induction on n.

- $n = 0$, $choiceMac(\alpha, A) = \emptyset$, the precondition is $false$, hence the whole proposition is $true$;

- $n = 1$, α is primitive action, every A satisfying the precondition is its nature's choice, then our proposition is same as (b), therefore the proposition is *true*;

- Now, we suppose the proposition is true for $n < k$ where k is some natural number, we are considering for $n = k$, i.e. for any $\alpha = \alpha_1; \alpha_2; \cdots; \alpha_k$, suppose there exists A^0 such that $A^0 = A^0_1; A^0_2; \cdots; A^0_k \wedge choiceMac(\alpha, A^0) \wedge Poss(A^0, s)$, we will prove that

$$\sum_{A \in maxPoss(\alpha,s)} probMac(A, \alpha, s) = 1.$$

Let $\alpha' = \alpha_1; \alpha_2; \cdots; \alpha_{k-1}$, since A^0 satisfying that $choiceMac(\alpha, A^0) \wedge Poss(A^0, s)$, therefore, let $A' = A^0_1; A^0_2; \cdots; A^0_{k-1}$, and it is easy to see that A' satisfies

$$choiceMac(\alpha', A') \wedge Poss(A', s).$$

By hypothesis,we have

$$\sum_{A \in maxPoss(\alpha',s)} probMac(A, \alpha', s) = 1.$$

Then for every $A \in maxPoss(\alpha', s)$, suppose $A = A_1; \cdots; A_m$ for some $m \leq k - 1$,

(1) if $m < k - 1$, then $A \in maxPoss(\alpha, s)$ by definition of $maxPoss$;

(2) if $m = k-1$ and for any A_k satisfying $choice(\alpha_k, A_k) \supset \neg Poss(A; A_k, s)$, then $A \in maxPoss(\alpha, s)$ by definition of $maxPoss$;

(3) if $m = k - 1$ and there exists a_k satisfying $choice(\alpha_k, A_k) \wedge Poss(A; A_k, s)$, then $Poss(A_k, do([A_1, \cdots, A_{k-1}], s))$ by the definition of $Poss$ for deterministic sequential action, therefore by induction assumption

$$\begin{aligned} &\sum_{b \in maxPoss(\alpha_k, do([A_1,\cdots,A_{k-1}],s))} probMac(b, \alpha_k, do([A_1, \cdots, A_{k-1}], s)) \\ = &\sum_{b \in maxPoss(\alpha_k, do([A_1,\cdots,A_{k-1}],s))} prob_0(b, \alpha_k, do([A_1, \cdots, A_{k-1}], s)) = 1 \end{aligned}$$

so, for fixed A, let $B'_i = A; B_i$ for every $B_i \in maxPoss(\alpha_k, do([A_1, \cdots, A_{k-1}], s))$, we have $B'_i \in maxPoss(\alpha, s)$, and

$$\begin{aligned} & \sum_i probMac(B'_i, \alpha, s)) \\ = \ & probMac(A, \alpha', s) * \\ & (\sum_{B \in maxPoss(\alpha_k, do([A_1, \cdots, A_{k-1}], s))} prob_0(B, \alpha_k, do([A_1, \cdots, A_{k-1}], s))) \\ = \ & probMac(A, \alpha', s). \end{aligned}$$

Therefore, by (1),(2), and (3), we have

$$\begin{aligned} & \sum_{A \in maxPoss(\alpha, s)} probMac(A, \alpha, s) \\ = & \sum_{A \in maxPoss(\alpha', s)} probMac(A, \alpha', s) = 1. \end{aligned}$$

Hence, we proved the proposition is true for all $n \in \mathcal{N}$. ■

This theorem indicates the common sense of the property of probabilities. The definition of $maxPoss$ and the property above are what we are really interested in. Because as we argued in the motivation, when the agent meets the macro-action in the same local situation it has met before, rather than recomputes the possible choices and probabilities, we would like the agent to "recall" the maximal possible deterministic sequences it has computed and remembered before.

Up till now, everything works properly for sequential actions, and it is reasonable for us to consider the *macro-action* can be of form $\alpha_1; \alpha_2; \cdots; \alpha_n$ for n stochastic actions where n is a finite natural number no less than 2.

Finally, there are still two complex actions we need to consider. We think it is not an obligation to keep the test action "?" as a part of macro-actions. The reason is that "?" is not actually an action which might affects the state of the environment, i.e., executing the test action will not affect the truth values of any fluents in the system. There is no extra intermediate information we need to keep for executing the test action. However, joining the testing actions into the macro-actions may make it difficult for us to define the nature's choices for a macro-action, since the tests can appear anywhere in the macro-action and these tests are situation-based. Similarly, for the conditional complex action $if\ \varphi\ then\ \alpha\ else\ \beta$ we are more interested in the probability results of executing of the body α or β than the testing of φ. Hence, currently we would like to keep our life easy and will not bind the testing and the conditional complex action into macro-actions. Maybe in the near future, we would like to discuss what will happen if we combine these two actions as parts of the macro-actions under certain condition.

3.2.2 Spotting Macro-action

What does spotting macro-actions mean? Why do we need to do this? Given a very long sequence of stochastic actions, for example,

$$go(office(Sue)); giveMail(Sue); giveCoffee(Sue); go(office(Pat));$$
$$giveMail(Pat); giveCoffee(Pat),$$

we are not willing to treat it as a whole macro-action, since it is actually can be considered as performing same macro-action $go(office(p)); giveMail(p); giveCoffee(p)$ on different instances, which has unique precondition axiom, successor state axiom and probability computing formulas. Moreover, if we do not have obvious disparity between normal sequential actions and the macro-actions, it is difficult for the agent to identify macro-actions from a long sequence of stochastic actions. Another reason is that the macro-actions actually can be viewed as a kind of special procedures, but we still need to differ it from the ordinary ones. Finally, if the macro-actions have names, it will be convenient for both the controller and the autonomous agent to remember and recognize them. Hence, similar to procedures, we introduce terms `macro` and `endmacro` such that

$$\texttt{macro}\ p_{name}\ \ \Delta\ \texttt{endmacro},$$

meaning that Δ is a macro-action named p_{name}.

Therefore, as an example, if we define

$$\texttt{macro}\ serve(p)\ \ go(office(p)); giveMail(p); giveCoffee(p)\ \texttt{endmacro},$$

then the example we mentioned at the beginning of this sub-section can be represented as

$$serve(Sue); serve(Pat).$$

All in all, we summarize the definition of *macro-action* as follows:

Definition 3.10 *A* macro-action *in an uncertainty system described is a sequence of stochastic actions* $\alpha_1; \alpha_2; \cdots ; \alpha_n$ *with some name* p_{name}*, denoted as*

$$\texttt{macro}\ p_{name}\quad \alpha_1; \alpha_2; \cdots ; \alpha_n\ \texttt{endmacro},$$

where $n \in \mathcal{N}$ *and* $n \geq 2$*. We also say that "*p_{name} is a macro-action with body $\alpha_1; \alpha_2; \cdots ; \alpha_n$*".*

To simplify the problem in this work, we do not allow the nesting of macro-actions, that is to say, the body of p_{name} only consists of sequence of stochastic actions. Moreover, the definitions of macro-actions in a dynamic system must follow the unique name axiom.

As a result, the definitions of $choiceMac$, $probMac$, $seqLength$ and $maxPoss$ in above sub-section also can be extended as follows.

Definition 3.11 *For any macro-action* p_{name} *with body* Δ*,*

$choiceMac(p_{name}, a) \equiv choiceMac(\Delta, a),$

$probMac(A, p_{name}, s) = p \equiv probMac(A, \Delta, s) = p,$

$seqLength(p_{name}) = seqLength(\Delta),$

$maxPoss(p_{name}, s) = l \equiv maxPoss(\Delta, s) = l.$

Moreover, for later convenience, we would like to give following description of notations.

Notation 3.12 *For any macro-action or sequential action α, let $\alpha[n]$ denote the n^{th} deterministic or stochastic action of α or of α's body for $1 \leq n \leq seqLength(\alpha)$, i.e.*

1. *if α is a macro-action with body Δ , then $\alpha[n] = \Delta[n]$;*
2. *otherwise, if $n \leq 0$ or $n > seqLength(\alpha)$, then $\alpha[n] \stackrel{def}{=} nil$, else $\alpha[n]$ is of sort action such that there exist actions $a_1, a_2, \cdots, a_{n-1}, a_{n+1}, \cdots, a_{seqLength(\alpha)}$ satisfying that*

$$\alpha = a_1; a_2 \cdots ; a_{n-1}; \alpha[n]; a_{n+1}; \cdots ; a_{seqLength(\alpha)}.$$

Notation 3.13 *Notice that, up till now, we extend several terms of the language $\mathcal{L}_{sc}$ described in Chapter 2.1 as follows:*

1. *the term $do(a, s)$ is extended to the form $do(a_1; a_2; \cdots ; a_n, s)(n \geq 1)$ where every a_i is of sort action and s is of sort situation;*
2. *the predicate $Poss(A, s)$ is extended to the form $Poss(A_1; A_2; \cdots ; A_n, s)(n \geq 1)$ stating that deterministic sequential action $A_1; A_2; \cdots ; A_n$ can be performed in the situation s.*

We denote language $\mathcal{L}_{sc}$ with above extensions as language $\mathcal{L}'_{sc}$, i.e., the difference between $\mathcal{L}_{sc}$ and $\mathcal{L}'_{sc}$ is that $\mathcal{L}_{sc}$ does not allow above two kinds of extended terms.

After observation and discussion above, we finally decided the structure of macro-action, and found that it is possible to develop a knowledge base for macro-action, which will include the extended successor state axioms, action preconditions, and probabilities for the macro-actions. We would like to see this work later in next chapter, and now we may look at following example of robot climbing stairs for better understanding.

3.3 Example of Macro-actions for Robot Climbing Stairs

Continuing Example 3.1 in previous section, suppose we have

$$\begin{aligned}
&\texttt{macro}\ \ stepMain(h) \\
&\quad liftUpperLeg(h); forwLowLeg; stepDown(main); moveBarycenter(main); \\
&\quad straightLeg \\
&\texttt{endmacro},
\end{aligned} \tag{3.3}$$

$$\begin{aligned}
&\texttt{macro}\ \ stepSupp \\
&\quad forwSupLeg; stepDown(supporting); moveBarycenter(supporting) \\
&\texttt{endmacro},
\end{aligned} \tag{3.4}$$

i.e., we define two macro-actions $stepMain(h)$ and $stepSupp$, and the procedure of climbing a stair of height h therefore can be defined as follows:

$$\texttt{proc}\ \ climbing(h)\quad ?(legalStair(h)); stepMain(h); stepSupp\ \ \texttt{endproc}. \tag{3.5}$$

We also can define following macro-action

$$\begin{aligned}
&\texttt{macro}\ \ climbStair(h) \\
&\quad liftUpperLeg(h); forwLowLeg; stepDown(main); moveBarycenter(main); \\
&\quad straightLeg; forwSupLeg; stepDown(supporting); moveBarycenter(supporting) \\
&\texttt{endmacro}
\end{aligned} \tag{3.6}$$

and therefore the previous procedure $climbing(h)$ (3.1) can be represented as

$$\texttt{proc}\ \ climbing(h)\quad ?(legalStair(h)); climbStair(h)\ \ \texttt{endproc}. \tag{3.7}$$

According to the definition of $choiceMac$, for example

$$choiceMac(malfunc(h), stepMain(h)) \equiv true,$$

$$choiceMac(liftTill(h); stepDown(main), stepMain(h)) \equiv false,$$

$$choiceMac(forwSupLegS; stepDownF(supporting);$$
$$moveBarycenterS(supporting), stepSupp) \equiv true.$$

Since $seqLength(stepMain(h)) = 5$ and $seqLength(stepSupp) = 3$, then we have nature's choices of length from 1 to 5, therefore we might need all the extended successor state axiom for sequences $a_1; \cdots; a_n$ where $n = 2, 3, 4, 5$. For instance, consider fluent *barycenter* and for variables a_1 and a_2, we have

$$
\begin{aligned}
& barycenter(l, do(a_1; a_2, s)) \ \equiv \ \mathcal{R}[barycenter(l, do([a_1, a_2], s))] \\
= \quad & \mathcal{R}[a_2 = moveBarycenterS(l) \vee barycenter(l, do(a_1, s)) \wedge \\
& \neg(\exists l')[a_2 = moveBarycenterS(l') \wedge l \neq l']] \\
\equiv \quad & a_2 = moveBarycenterS(l) \vee \{a_1 = moveBarycenterS(l) \\
& \vee barycenter(l, s) \wedge \neg(\exists l')[a_1 = moveBarycenterS(l') \wedge l \neq l']\} \\
& \wedge \neg(\exists l')[a_2 = moveBarycenterS(l') \wedge l \neq l']].
\end{aligned}
$$

For other fluents and different lengths of deterministic sequential actions, the regression calculations are similar according to the description we gave in the previous section.

As examples of the extended action preconditions, both $liftTill(h); forwLowlLegS$ and $malfunc(h); forwLowlLegS$ are nature's choices of macro-action $stepMain(h)$, their precondition axioms are

$$
\begin{aligned}
& Poss(liftTill(h); forwLowlLegS, s) \\
\equiv \quad & Poss(liftTill(h), s) \wedge Poss(forwLowlLegS, do(liftTill(h), s)) \\
\equiv \quad & \mathcal{R}[Poss(liftTill(h), s) \wedge Poss(forwLowlLegS, do(liftTill(h), s))] \\
= \quad & \mathcal{R}[barycenter(supporting, s) \wedge \neg mainToCurr(wrongPos, do(liftTill(h), s)) \\
& \wedge \neg footOnGround(main, do(liftTill(h), s))] \\
\equiv \quad & barycenter(supporting, s) \text{ (by using regression and simplification), and}
\end{aligned}
$$

$$\begin{aligned}
&Poss(malfunc(h); forwLowlLegS, s)\\
&\equiv \mathcal{R}[Poss(malfunc(h), s) \wedge Poss(forwLowlLegS, do(malfunc(h), s))]\\
&= \mathcal{R}[barycenter(supporting, s) \wedge \neg mainToCurr(wrongPos, do(malfunc(h), s))\\
&\quad \wedge \neg footOnGround(main, do(malfunc(h), s))]\\
&= \mathcal{R}[barycenter(supporting, s) \wedge \neg true] \text{ (by regression and simplification)}\\
&\equiv false.
\end{aligned}$$

Other extended action preconditions can be obtained similarly. At last, some examples of the extended probabilities are given as follows according to the descriptions in the previous section:

i. for $A = liftTill(h); forwLowlLegS$ and $\alpha = stepMain(h)$, we have

$$\begin{aligned}
&probMac(liftTill(h); forwLowlLegS, stepMain(h), s) = p\\
&\equiv choiceMac(\alpha, A) \wedge Poss(A, s) \wedge p = prob_0(liftTill(h), liftUpperLeg(h), s)*\\
&\quad prob_0(forwLowlLegS, forwLowlLeg, do(liftTill(h), s))\\
&\quad \vee(\neg choiceMac(\alpha, A) \vee \neg Poss(A, s)) \wedge p = 0.
\end{aligned}$$

ii. $probMac(malfunc(h); forwLowlLegS, stepMain(h), s) = 0$, since we have the fact that $\neg Poss(malfunc(h); forwLowlLegS, s)$ for any situation s;

In this chapter, we discussed the motivation of the work that goes on in this paper. Next, we began with the first step of finding what we mean *macro-action* and argued that the structure is possible and reasonable for later development. finally, we worked on an concrete example of macro-actions to get more sense. Based on this, in next chapter, we are going to perform the formal work how to develop a database for macro-actions in an uncertainty model for the purpose of reuse in later application.

Chapter 4

Developing the Knowledge Base for Macro-actions

We have decided the frame of the macro-actions in last chapter. Now, we are going to introduce a knowledge base which stores the information of macro-actions of an uncertainty system. To develop the knowledge base, we present an algorithm for this procedure and implement it in Prolog.

4.1 The Components of the Knowledge Base

As we have seen the example of robot climbing stairs, the purpose of having a knowledge base for macro-actions is that the autonomous agent can reuse local information of macro-actions when it repeats the same procedures or strategies which are composed of macro-actions and other complex actions under the same state of environment at different time. The reason that we call it a *knowledge base* rather than *database* is that we not only want to save the results of extended probabilities of the nature's choices, but also want to keep the extended action axioms which are more like knowledge than data.

Suppose the controller presents the basic action theories $\mathcal{D}$ (including extended parts such as nature's choices and probabilities) for an uncertainty system as we described in Section 2.5, and wants to introduce several macro-actions $p_1, p_2, \cdots, p_t$ $(t \in \mathcal{N})$. We would like to develop a knowledge base for these macro-actions which consists of two parts: static part and dynamic part. People may ask why we need two parts and what exactly they look like. We feel it will be much easier for us to set forth the reasons after expressing the explicit components of these two parts than to argue the reasons first.

The static part of the knowledge base is as follows:

- Definitions of Macro-actions

  It consists of the statements of macro-action of the form

$$\texttt{macro}\ p_{name} \quad \Delta\ \texttt{endmacro}.$$

  We also introduce a special predicate $currentMaxLength(n)$ which denotes the maximal length of all the macro-actions in current knowledge base. Initially, when knowledge base is empty, we have the fact $currentMaxLength(0)$. Formally, it is defined as follows

$$\begin{aligned}
&currentMaxLength(n) \stackrel{def}{=} \\
&\quad if \text{ there is some macro-action in current knowledge base, } then \\
&\quad\ \ n = seqLength(p_0) \text{ for some macro-action (procedure) } p_0 \\
&\quad\ \ \text{in current knowledge base and } n \geq seqLength(p) \\
&\quad\ \ \text{for every macro-action } p \text{ in current knowledge base;} \\
&\quad else\ n = 0.
\end{aligned} \tag{4.1}$$

  Since we may have nature's choices of macro-actions ranging from length 1 to n satisfying that $currentMaxLength(n)$, we would like to keep the extended successor state axioms for deterministic sequences from length 1 to n in next part.

- Derived Theories for Macro-actions

  All the derived axioms are computed and stored here, for instance, preconditions axioms for every deterministic choices of macro-actions,etc. It includes the following three sub-groups.

  a. The Extended Successor State Axioms for Fluents

  Assume the maximum length of macro-actions declared in the first part is n, i.e., $currentMaxLength(n)$, we keep the extended successor state axioms of every fluent for deterministic sequences no longer than n as the form of

$$F(\vec{x}, do(a_1; a_2; \ldots; a_m, s)) \equiv \Psi_F(\vec{x}, a_1, a_2, \ldots, a_m, s), \tag{4.2}$$

  or

$$f(\vec{x}, do(a_1; a_2; \ldots; a_m, s)) = y \equiv \Psi_f(\vec{x}, y, a_1, a_2, \ldots, a_m, s), \tag{4.3}$$

  where m is natural number such that $2 \leq m \leq n$, F represents a relational fluent, f represents a functional fluent, and Ψ_F (respectively, Ψ_f) is some formula uniform in s obtained by regression and simplification.

  b. The Extended Precondition Axioms for Nature's Choices of Macro-actions

  As discussed in Section 3.2, we are interested in the extended precondition axioms for nature's choices of macro-actions. We keep them certainly for the sake of reuse. These extended Precondition Axioms are of form

$$Poss(A_1; A_2; \ldots; A_m, s) \equiv \Pi(\vec{t}, s), \tag{4.4}$$

  where $A_1; A_2; \ldots; A_m$ is a nature's choice of some macro-action declared in part a., and Π is some formula uniform in s obtained by regression and simplification. Moreover, since later we will implement the knowledge base by using Prolog, according to the property of the closed world assumption(CWA) [26], we need not keep those axioms in which Π's are $false$.

Additionally, we introduce a predicate $localChoice(\alpha, L)$, meaning that L is a list of all the nature's choices of macro-action α satisfying that

for any $A \in L$, there is precondition axiom of form $Poss(A, s) \equiv \Pi_A(s)$ either in this knowledge base or in $\mathcal{D}$,

i.e. we discard all those nature's choices of a macro-action that are obviously not possible to be performed in any situation. We gather such information for every existing macro-action in this knowledge base. To do this additional information collecting operation can bring us the advantage that the agent later won't waste time on those non-performable choices.

c. The Extended Probabilities for Nature's Choices of Macro-actions

The most important information for uncertainty system is the probability of every nature's choice of macro-action, which is as the given definition of $probMac$ in Section 3.2. To achieve the goal of reusing useful results of macro-actions rather than recomputing them, we prefer saving the regression results which are uniform in s for the definition of $probMac(A, \alpha, s)$. But, since we have already had the information of the extended precondition axioms kept in the knowledge base, rather than using the original definition in Chapter 3, the following equivalent definition is more suitable for us (the equality can be easily proved by induction):

given A representing deterministic sequential action and variable α representing a stochastic action or a macro-action,

$$\begin{aligned} &probMac(A, \alpha, s) = p \stackrel{def}{=} \\ &\quad choiceMac(\alpha, A) \wedge Poss(A, s) \wedge p = probMac_0(A, \alpha, s) \\ &\quad \vee(\neg choiceMac(\alpha, A) \vee \neg Poss(A, s)) \wedge p = 0, \end{aligned} \tag{4.5}$$

in which we introduce the supplementary predicate $probMac_0(A, \alpha, s)$ defined recursively as follows:

$$
\begin{aligned}
& probMac_0(A,\alpha,s) = p \stackrel{def}{=} \\
& \quad \text{if } seqLength(A) = 1 \text{ then } p = prob_0(A, \alpha[1], s), \\
& \quad \text{else } (\exists y_1, y_2)[y_1 = probMac_0(A[1]; \cdots ; A[n-1], \alpha, s) \wedge \\
& \qquad y_2 = prob_0(A[n], \alpha[n], do(A[1]; \cdots ; A[n-1], s)) \wedge p = y_1 * y_2] \\
& \qquad \text{where } n = seqLength(A) \wedge n > 1.
\end{aligned} \tag{4.6}
$$

Notice that if we have $Poss(A, s) \equiv false$ for some macro-action α's choice A, i.e. there is no rule for $Poss(A, s)$ in part b., by CWA [26] and negation as failure, we could definitely know that $probMac(A, \alpha, s) = 0$ by definition (4.5), therefore, it is not necessary to compute and save the regression result of $probMac_0(A, \alpha, s)$ for such A. Hence, we only need to keep the regression results of $probMac_0$ as follows

$$probMac_0(A, \alpha, s) = p \quad \equiv \quad f(p, \vec{t}, s) \tag{4.7}$$

where f is a formula uniform in s obtained by regression and simplification of (4.6) for every α's nature's choice A which has the precondition axiom in $\mathcal{D}$ or in part **b.**.

We have finished describing the components of the static part of the knowledge base. Clearly, all of the knowledge we keep in this part are universal in the sense that they are not relevant to particular situations, i.e., s is a variable of sort situation and we can obtain the knowledge for macro-actions described above without any descriptions of the initial database and any exact programs. Therefore, "static" does not mean that this part could not change at all, it means that the static part of a knowledge base is relatively stable and will not change with the changing of the initial database and programs. Controller can extend this part by adding more macro-actions, or totally discard the whole part by deleting the file that is used to save the above information and re-build a new one.

The dynamic part of the knowledge base depends on particular situations. This part contains 3-ary predicate $maxPossBase$ facts such that

$$maxPossBase(List, \alpha, S) \equiv List = maxPoss(\alpha, S)$$

for some macro-action α and situation instance S. The information of $maxPoss(\alpha, S)$ we discussed in previous chapter is very useful and the reuse of it can save computational time for the autonomous agent when it performs the macro-action (possibly on different instances) under the same situation. These facts, $maxPossBase(List, \alpha, S)$, depend on particular situations, therefore relate to the initial database and programs. They are generated during executing and will disappear when the controller reloads new initial database. Since this dynamic part is related to the initial situation, we embed the generation into application interpreter. The detail will be discussed later during application in Chapter 5.

By giving the descriptions of the knowledge base, it is clear that why we would like to separate it into two parts. We would not like the general knowledge in the static part to disappear so easily, while, on the other hand, would not like to keep the situation instance related information any more once the initial database changes.

4.2 An Extended Regression Operator Based on the Knowledge Base

Given the structures of knowledge base for macro-actions, it is very natural for us to think of introducing an extended regression operator, which will help us develop the knowledge base formally and later for the purpose of reusing the extended axioms in the base. Our new regression operator $\mathcal{R}^\star$ will be defined on s-regressable formula in $\mathcal{L}'_{sc}$, i.e., we allow formulas to include the extended terms described in Notation 3.13.

Definition 4.1 *Suppose s either is the initial situation S_0 or a variable of sort situation. A formula W of $\mathcal{L}'_{sc}$ is s-*regressable *for some situation s iff*

1. *every term of sort situation mentioned by W has the form $do(\alpha_n, \cdots, do(\alpha_1, s)\cdots)$ for some $n \geq 0$, and every α_i $(1 \leq i \leq n)$ either is of sort action or is of form $\alpha_{i,1}; \cdots ; \alpha_{i,m_i}$ for some $m_i \geq 2$ and every $\alpha_{i,j}$ $(1 \leq j \leq m_i)$ is of sort action;*

2. *for every atom of the form $Poss(\alpha, \sigma)$ mentioned by W, $\alpha = A_1(\vec{x_1}); \cdots ; A_n(\vec{x_n})$ for some $n \geq 1$ and all A_i are action function symbols of $\mathcal{L}_{sc}$;*

3. *other conditions are same as the 3^{rd} and 4^{th} conditions in the Definition 2.3.*

*A functional fluent term is s-*prime *for s, iff it has the form $f(\vec{t}, do(\alpha_n, \cdots, do(\alpha_1, s)\cdots)$ for $n \geq 1$, where every α_i $(1 \leq i \leq n)$ either is of sort action, or is of form $\alpha_{i,1}; \cdots ; \alpha_{i,m_i}$ for some $m_i \geq 2$ and every $\alpha_{i,j}$ $(1 \leq j \leq m_i)$ is of sort action; and each of the terms $\vec{t}$ is uniform in s.*

And now, we give the definition of the extended regression operator $\mathcal{R}^\star$ for s-regressable formula W in $\mathcal{L}'_{sc}$ (where s is either S_0 or a variable of sort situation) as follows:

Definition 4.2 The Extended Regression Operator

1. *Suppose $W = Poss(\alpha(\vec{t}), \sigma)$ where $\alpha(\vec{t})$ is a sequence of deterministic actions (including of length 1, i.e., primitive action) and σ is of sort situation, there are two cases:*

   (a) *If there is (extended) action precondition axiom given as*

   $$Poss(\alpha(\vec{x}), s_1) \equiv \Pi_\alpha(\vec{x}, s_1),$$

   without loss of generality, assume that all quantifiers (if any) of $\Pi_\alpha(\vec{x}, s_1)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $Poss(\alpha(\vec{t}), \sigma)$, then

   $$\mathcal{R}^\star[W] \quad = \quad \mathcal{R}^\star[\Pi_\alpha(\vec{t}, \sigma)].$$

(b) *Otherwise, we must have* $seqLength(\alpha(\vec{t})) > 1$*, and suppose the recursive definition of* $Poss(a_1; \cdots ; a_n, s)$ *for* $n > 1$ *is of form*

$$\begin{aligned} &Poss(a_1; \cdots ; a_n, s) \equiv \\ &\quad Poss(a_1; \cdots ; a_{n-1}, s) \wedge Poss(a_n, do(a_1; \cdots ; a_{n-1}, s)), \end{aligned} \tag{4.8}$$

which is equivalent to the original definition formula(3.2) (can be proved easily), without loss of generality, assume that all quantifiers (if any) of above formula have had their quantified variables renamed to be distinct from the free variables (if any) of $Poss(\alpha(\vec{t}), \sigma)$*, then let*

$$\begin{aligned} \mathcal{R}^{\star}[W] \;=\; & \mathcal{R}^{\star}[Poss(\alpha_1(\vec{t_1}); \cdots ; \alpha_{n-1}(\vec{t_{n-1}}), \sigma) \wedge \\ & Poss(\alpha_n(\vec{t_n}), do((\alpha_1(\vec{t_1}); \cdots ; \alpha_{n-1}(\vec{t_{n-1}}), \sigma))]. \end{aligned}$$

2. *Suppose* W *is a s-regressable atom, but not a* $Poss$ *atom. There are three possibilities:*

(a) s *is the only term of sort situation (if any) mentioned by* W*, then*

$$\mathcal{R}^{\star}[W] \;=\; W.$$

(b) *Suppose that* W *mentions a term of the form* $g(\vec{t}, do(\alpha', \sigma'))$ *for some functional fluent* g*,* $\alpha' = \alpha'_1; \cdots ; \alpha'_n$ *for some* $n > 0$ *and every* α'_i *is of sort action, and* σ' *is of sort situation.* $g(\vec{t}, do(\alpha', \sigma'))$ *mentions a s-prime functional fluent term of form* $f(\vec{r}, do(\alpha, \sigma))$ *where* $\alpha = \alpha_1; \cdots ; \alpha_m$ *for some* $m > 0$ *and every* α_i *is of sort action, and* σ *is of sort situation.*

- *If there is formula of form*

$$f(\vec{x}, do(a_1; \cdots ; a_m, s_1)) = y \equiv \psi_f(\vec{x}, y, a_1, \cdots , a_m, s_1)$$

in the knowledge base, without loss of generality, assume that all quantifiers (if any) of $\psi_f(\vec{x}, y, a_1, \cdots , a_m, s_1)$ *have had their quantified variables* *renamed to be distinct from the free variables (if any) of* $f(\vec{r}, do(\alpha, \sigma))$*, then*

$$\mathcal{R}^\star[W] \;=\; \mathcal{R}^\star[(\exists y).\psi_f(\vec{r}, y, \alpha_1, \cdots, \alpha_m, \sigma) \wedge W|_y^{f(\vec{r}, do(\alpha,\sigma))}];$$

- *otherwise, suppose* f*'s successor state axiom in* $\mathcal{D}_{ss}$ *is*

$$f(\vec{x}, do(a, s_1)) = y \equiv \phi_f(\vec{x}, y, a, s_1),$$

without loss of generality, assume that all quantifiers(if any) of $\phi_f(\vec{x}, y, a, s_1)$ *have had their quantified variables renamed to be distinct from the free variables (if any) of* $f(\vec{r}, do((\alpha, \sigma))$*, then let* $\sigma_1 = do(\alpha_1; \cdots; \alpha_{m-1}, \sigma)$ *(when* $m - 1 = 0$*,* $\sigma_1 = \sigma$*) and*

$$\mathcal{R}^\star[W] \;=\; \mathcal{R}^\star[(\exists y).\phi_f(\vec{r}, y, \alpha_m, \sigma_1) \wedge W|_y^{f(\vec{r}, do(\alpha,\sigma))}].$$

Here y *is a variable not occurring free in* $W, \vec{r}, \alpha$ *or* σ*.*

(c) W *is a relational fluent atom of form* $F(\vec{t}, do(\alpha, \sigma))$ *where* $\alpha = \alpha_1; \cdots; \alpha_n$ *for* $n > 0$ *and every* α_i *is of sort action, and* σ *is of sort situation.*

- *If there is formula of form*

$$F(\vec{x}, do(a_1; \cdots; a_n, s_1)) \equiv \psi_F(\vec{x}, a_1, \cdots, a_n, s_1)$$

in the knowledge base, without loss of generality, assume that all quantifiers (if any) of $\psi_F(\vec{x}, a_1, \cdots, a_n, s_1)$ *have had their quantified variables renamed to be distinct from the free variables (if any) of* $F(\vec{t}, do(\alpha, \sigma))$*, then*

$$\mathcal{R}^\star[W] \;=\; \mathcal{R}^\star[\psi_F(\vec{t}, \alpha_1, \cdots, \alpha_n, \sigma)];$$

- *otherwise, suppose* F*'s successor state axiom in* $\mathcal{D}_{ss}$ *is*

$$F(\vec{x}, do(a, s_1)) \equiv \Phi_F(\vec{x}, a, s_1),$$

without loss of generality, assume that all quantifiers(if any) of $\Phi_F(\vec{x}, a, s_1)$ *have had their quantified variables renamed to be distinct from the free variables (if any) of* $F(\vec{t}, do(\alpha, \sigma))$*, then let* $\sigma_1 = do(\alpha_1; \cdots; \alpha_{n-1}, \sigma)$ *(when* $n - 1 = 0$*,* $\sigma_1 = \sigma$*) and*

$$\mathcal{R}^\star[W] \;=\; \mathcal{R}^\star[\Phi_F(\vec{t}, \alpha_n, \sigma_1)].$$

3. *For non-atomic formulas, regression is defined inductively as follows.*

$$\mathcal{R}^\star[\neg W] = \neg\mathcal{R}^\star[W]$$
$$\mathcal{R}^\star[W_1 \wedge W_2] = \mathcal{R}^\star[W_1] \wedge \mathcal{R}^\star[W_2]$$
$$\mathcal{R}^\star[(\exists x)W] = (\exists x)\mathcal{R}^\star[W]$$

Because regression repeatedly substitutes logically equivalent formulas for atoms, what the operator delivers will be logically equivalent for what it starts with. This forms the basis of the following:

Theorem 4.3 *Suppose* W *is a s-regressable sentence of* $\mathcal{L}'_{sc}$ *for some situation* s *that mentions no functional fluents, and* $\mathcal{D}$ *is a basic theory of actions. Then* $\mathcal{R}^\star$ *is a sentence uniform in* s*. Moreover,*

$$\mathcal{D} \models W \equiv \mathcal{R}^\star[W].$$

According to above theorem and Theorem 4.5.1, Theorem 4.5.2 in [28], we also have the following properties:

Theorem 4.4 *Suppose* W *is a regressable sentence of* $\mathcal{L}_{sc}$ *that mentions no functional fluents, and* $\mathcal{D}$ *is a basic theory of actions. Then*

$$\mathcal{D} \models \mathcal{R}[W] \equiv \mathcal{R}^\star[W].$$

Theorem 4.5 *Suppose W is a regressable sentence of $\mathcal{L}_{sc}$ that mentions no functional fluents, and $\mathcal{D}$ is a basic theory of actions. Then*

$$\mathcal{D} \models W \; iff \; \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}^{\star}[W].$$

Moreover, as we discussed in previous chapter, $do(a_1; a_2; \cdots; a_n, s)$ represents the same situation as $do([a_1, a_2, \cdots, a_n], s)$ for deterministic actions. Hence, for any S_0-regressable formula W_1 in $\mathcal{L}'_{sc}$, there is a regressable formula W_2 equivalent to W_1 obtained by replacing any $Poss(\alpha, \sigma)$ in W_1 with its equivalent formula of form (3.2) if $seqLength(\alpha) > 1$ and replacing any $do(a_1; a_2; \cdots; a_n, \sigma)$ in W_1 with $do([a_1, a_2, \cdots, a_n], \sigma)$. We call W_2 as the *equal formula of* W_1 *in* $\mathcal{L}_{sc}$, and it is easy to see that

Theorem 4.6 *Suppose W_1 is a S_0-regressable sentence of $\mathcal{L}'_{sc}$ that mentions no functional fluents, W_2 is the equal formula of W_1 in $\mathcal{L}_{sc}$, and $\mathcal{D}$ is a basic theory of actions. Then*

$$\mathcal{D} \models W_2 \; iff \; \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}^{\star}[W_1].$$

These mean that our regression operator $\mathcal{R}^{\star}$ can obtain the equivalent result as the original operator. Based on the definition of regression operator $\mathcal{R}^{\star}$ and above properties, we are now going to develop the knowledge base developer which is a program used to develop the static part of the knowledge base in a formal way.

4.3 The Knowledge Base (Static Part) Developer

We have seen how we designed the knowledge base, and we are going to present how the agent develops the knowledge base (static part) based on the user-provided basic action theories and descriptions of macro-actions. Our basic idea is that for a given dynamic system with its knowledge base, if the controller wants to use some macro-actions that are not in the base, he or she will call a program built in the agent to compute and add the information of the new macro-actions into the static part of the knowledge base before using them; otherwise, the controller can use the existing macro-actions directly.

4.3.1 The Algorithm

Suppose the controller has provided the description of the basic action theories $\mathcal{D}$ together with declarations of nature's choices and probabilities $prob_0$ for an uncertainty system M. Notice that the initial knowledge base of macro-actions for M is empty, since there is no macro-action being declared yet. We claim that the declaration of macro-actions in the knowledge base is not opaque to the controller, i.e., the controller can easily retrieve how many macro-actions have been declared in current base and how they are defined. If the controller thinks it is necessary to introduce new macro-actions, it is the controller's duty to give the names and corresponding bodies of the new macro-actions.

The knowledge base developer $kbDeveloper(list, base)$ is a program used by the agent to compute and add information into static part of knowledge base $base$ for new macro-actions in $list$ provided by the controller. The detailed algorithm of the developer $kbDeveloper(list, base)$ is as follows:

The Algorithm of Developing a Knowledge Base(Static Parts)

1. Let $n1$ be the number satisfying $currentMaxLength(n1)$, i.e. we update $currentMaxLength$ everytime we call this program.
2. Let $n2$ be the maximal length of the macro-actions in $list$.
3. (adding declarations of new macro-actions)

   Insert declarations of new macro-actions in $list$ into knowledge base and current program.

4. (adding the successor state axioms)

   If $n2 \leq n1$, then go to next step;

   else, do following step:

   for $i = n1 + 1$ to $n2$ do

   (1) for every relational fluent $F(\vec{x}, s)$, compute $\mathcal{R}^{\star}[F(\vec{x}, do(a_1; \cdots ; a_i, s))]$ and get result $\phi_{F,i}(\vec{x}, a_1, \cdots, a_i, s)$ for some $\phi_{F,i}$ uniform in s and insert formula

   $$F(\vec{x}, do(a_1; \cdots ; a_i, s)) \equiv \phi_{F,i}(\vec{x}, a_1, \cdots, a_i, s)$$

   into the knowledge base and the front of current program;

   (2) similarly, compute $\mathcal{R}^{\star}[f(\vec{x}, do(a_1; \cdots ; a_i, s)) = y]$ for every functional fluent $f(\vec{x}, s)$ and get result $\phi_{f,i}(\vec{x}, y, a_1, \cdots, a_i, s)$ for some $\phi_{f,i}$ uniform in s and insert formula

   $$f(\vec{x}, do(a_1; \cdots ; a_i, s)) = y \equiv \phi_{f,i}(\vec{x}, y, a_1, \cdots, a_i, s)$$

   into the knowledge base and the front of current program.

5. (adding the extended action preconditions for nature's choices of new macro-actions)

   For every macro-action α given in *list* do

   for $j = 2$ to $seqLength(\alpha)$ do

   compute $\mathcal{R}^{*}[Poss(A(\vec{t}), s)]$ for every deterministic sequential actions $A(\vec{t})$ satisfying that $choiceMac(\alpha, A(\vec{t}))$ and $seqLengh(A(\vec{t})) = j$ to get the result $\Pi_A(\vec{t}, s)$; if it is not equal to $false$, then insert formula

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

   into the knowledge base and the front of current program.

6. gather facts $localChoice(\alpha, L)$ for every α in $list$ and insert them into the knowledge base and current program.

7. (adding $probMac_0$ for nature's choices of new macro-actions)
For every macro-action α given in $list$ do
for every $A(\vec{t})$ in L satisfying $localChoice(\alpha, L)$, compute $probMac_0(A(\vec{t}), \alpha, s)$ recursively according to the definition of formula (4.6) as follows:
If $seqLength(A(\vec{t})) = 1$, then

$$probMac_0(A(\vec{t}), \alpha, s) = f_A(\vec{t}, s),$$

where $f_A(\vec{t}, s) = prob_0(A(\vec{t}), \alpha[1], s)$,
else if $A(\vec{t}) = A_1; A_2; \cdots; A_n$ $(n > 1)$ and we have computed

$$probMac_0(A_1; A_2; \cdots; A_{n-1}, \alpha, s) = y_1 \equiv f'(\vec{t}, y_1, s)$$

where f' is uniform in s, then

$$probMac_0(A(\vec{t}), \alpha, s) = y \equiv y = y_1 * y_2 \wedge f'(\vec{t}, y_1, s) \wedge f_{A_n}(\vec{t}, y_2, s),$$

where $f_{A_n}(\vec{t}, y_2, s)$ is obtained by computing $\mathcal{R}^*[y_2 = W]$ in which W is the equivalent formula of $prob_0(A_n, \alpha[n], do(A_1; A_2; \cdots; A_{n-1}, s))$ according to the definition of $prob_0$. We therefore get an equivalent formula $f(\vec{t}, y, s)$ of $probMac_0(A(\vec{t}), \alpha, s) = y$ uniform in s, and then insert $probMac_0(A(\vec{t}), \alpha, s) = y \equiv f(\vec{t}, y, s)$ into the knowledge base and current program.

The reason for using extended regression operator is obvious. For example, to compute the regression result of $F(\vec{x}, do(a_1; a_2; \cdots; a_n, s))$ $(n > 1)$ given all the formulas of (4.3) and (4.4) for $a_1; a_2; \cdots; a_{n-1}$ exist, we only need two steps by using operator $\mathcal{R}^*$, but need n steps by using operator $\mathcal{R}$.

4.3.2 Implementation and Experiment

We implement the algorithm in Prolog which is the language that we are used to implement Golog and stGolog of the action theory.

First thing we need to do is that to choose a proper structure to represent deterministic sequential actions. Notice that the structure "list" in Prolog is similar to definition of *log* and easy to deal with; moreover, we never used this structure before in the implementation of uncertainty system therefore should not cause conflict. Hence, in implementation of the work in this paper, we will use lists to represent the nature's choices of macro-actions, and always use form $do([a_1, a_2, \cdots, a_n], s)$ $(n \geq 1)$ to represent the situation $do(a_1; a_2; \cdots; a_n, s)$. Second, because we need to compute those extended axioms by using regression, the controller needs to provide the successor state axioms, precondition axioms and probabilities $prob_0$ in the form of `Atom <=> Expression`. For example, in the system of robot climbing stairs, controller provides the following assertions:

Action Precondition and Successor State Axioms for the Developer

```
poss(liftTill(H),S) <=> barycenter(supporting,S).

poss(malfunc(H),S) <=> barycenter(supporting,S).

poss(forwLowLegS,S) <=> -mainToCurr(wrongPos,S) & -footOnGround(main,S).

poss(forwLowLegF,S) <=> -mainToCurr(wrongPos,S) & -footOnGround(main,S).

poss(stepDownS(L),S) <=> -footOnGround(L,S) & overNewStair(L,S).

poss(stepDownF(L),S) <=> -footOnGround(L,S) & overNewStair(L,S).

poss(moveBarycenterS(L),S) <=> footOnGround(L,S).

poss(moveBarycenterF(L),S) <=> footOnGround(L,S).

poss(straightLeg,S) <=> -straightMain(S) & footOnGround(main,S) &
                        barycenter(main,S).

poss(forwSupLegS,S) <=> barycenter(main,S) & straightMain(S).
```

```
poss(forwSupLegF,S) <=> barycenter(main,S) & straightMain(S).


straightMain(do([A],S)) <=> A = straightLeg v
                            straightMain(S) & -(A = liftTill(H)).
barycenter(L,do([A],S)) <=> A = moveBarycenterS(L) v
           barycenter(L,S) & -(A = moveBarycenterS(L1) & -(L=L1)).
footOnGround(L,do([A],S)) <=> A = stepDownS(L) v
           footOnGround(L,S) & (L = main & -(A = liftTill(H)) v
           L = supporting & -(A = straightLeg)).
overNewStair(L,do([A],S)) <=> A = forwLowLegS & L = main v
           A = forwSupLegS & L = supporting v
           overNewStair(L,S) & -(A = stepDownS(L)).
mainToCurr(H,do([A],S)) <=>  A = malfunc(H1) & H = wrongPos v
           A = liftTill(H) v A = stepDownS(main) & H = 0 v
           mainToCurr(wrongPos,S) & H = wrongPos v
           mainToCurr(H,S) & -(H = wrongPos) & -(A= malfunc(H1)) &
           -(A = liftTill(H1) & -(H = H1)) & -(A = stepDownS(main)).


% Probabilities
prob0(liftTill(H),liftUpperLeg(H),S,Pr) <=> Pr is 100/(H+100).
prob0(malfunc(H),liftUpperLeg(H),S,Pr) <=> Pr is H/(H+100).
prob0(forwLowLegS,forwLowLeg,S,Pr) <=> mainToCurr(H,S) & Pr is 80/(H+80).
prob0(forwLowLegF,forwLowLeg,S,Pr) <=> mainToCurr(H,S) & Pr is H/(H+80).
prob0(stepDownS(L),stepDown(L),S,Pr) <=> Pr = 0.9.
prob0(stepDownF(L),stepDown(L),S,Pr) <=> Pr = 0.1.
prob0(moveBarycenterS(L),moveBarycenter(L),S,Pr) <=> Pr = 0.8.
prob0(moveBarycenterF(L),moveBarycenter(L),S,Pr) <=> Pr = 0.2.
prob0(straightLeg,straightLeg,S,Pr) <=> Pr = 1.0.
prob0(forwSupLegS,forwSupLeg,S,Pr) <=> Pr = 0.8.
```

```
prob0(forwSupLegF,forwSupLeg,S,Pr) <=> Pr = 0.2.
```

Notice that in the successor state axioms, we modify do(A,S) to be do([A],S), which can make the agent realize that A is variable of primitive action instead of a general variable which may cause mistakes during regression.

Besides these assertions, the user also needs to provide the declarations of the nature's choices of stochastic actions as of form `choice(a,C):- C=a1;C=a2;`$\cdots$`;C=a`m. Moreover, according to the algorithm, we need to gather all the fluents and construct new head of the extended successor state axioms. Notice that there is an existing term `restoreSitArg(F,S,F[S])` ([28] Chapter6.3.2) introduced in the very beginning of the implementation of Golog meaning that the result of restoring the situation `S` to the situation-suppressed fluent atom `F` is `F[S]`. We always have a collection of clauses of such form for all the fluents in the system. We therefore can use this collection to help us find all fluents we need to work on as well as to generate the head of the new extended successor state axioms for different situations $do([A_1, A_2, \cdots, A_n], S)$. Hence, the aspects described above are all the clauses we need to provide for generating a new knowledge base (static part) for an uncertainty system. If we have had a knowledge base (static part) for the system and want to extend it with new macro-actions, we also need to provide all the extended successor state axioms in it for the purpose of avoiding duplication and of saving time. Notice that we will use files to store descriptions of the systems and the knowledge base (static part), therefore, the assertions of these rules can be done by simply loading the corresponding files.

There is another problem we need to take care of. Since later we would rather to use Prolog clause form, `Head:- Body`, for the systems and knowledge base in the application, which is different from the form, `Head <=> Body`, we need for regression, hence we here desire to use two files, say *base1* and *base2*, to store the knowledge base (static part).

In *base1* we keep the declarations of the macro-actions and the extended successor state axioms in the form of `Head <=> Body`. When we need to the extend knowledge base (static part) by calling the developer, *base1* will be loaded. Moreover, in case that some nature's choice of the macro-actions is also other macro-action's nature's choice and we do not want to compute the extended action precondition for it again, we introduce a predicate `computedBefore(L)` to collect the deterministic sequence A for which we have computed $Poss(A, S)$ in every round of calling *kbDeveloper* into `L` and store it in *base1*, therefore avoid duplication later when we call *kbDeveloper* again for extending this system's knowledge base. In *base2*, all the components of knowledge base (static part) we declared in Section 4.1 are stored in the form of `Head:- Body`, and during application this file will be loaded in stead of *base1*. As a sequence, our *kbDeveloper* is modified from two arguments to be three arguments as *kbDeveloper(List, Base1,Base2)* where *List* is a list of new macro-actions in the Prolog list form `[name,body]`.

During the implementation, we break the algorithm into three parts: the main procedure, the regression procedures and the printing procedures (cf. Figure 4.2). And, the detailed program for `kbDeveloper/3` can be found in Appendix C. Most of the clauses in the program are self-explained. Only a few of them need some explanation as follows:

- quant(Head,Body,L): find quantified variables in the rules, which are actually the different variables in the body of the rule from the variables in the head.
- genNew(V,N,New): if N is a number, then New is a list of N different strings with prefix V; else if N and V are lists, New is a list of new strings such that the i^{th} element has a prefix if the i^{th} element of V and is different from all the variables in N.

After having the program, we take a look at the experiment on the example of robot climbing stairs. Suppose the description of system provided by the controller is saved as file *baseClimb*, the program of the developer is named *developer*, the knowledge base for development and later extension of knowledge base (as *base1* above) is named *climbBase1*

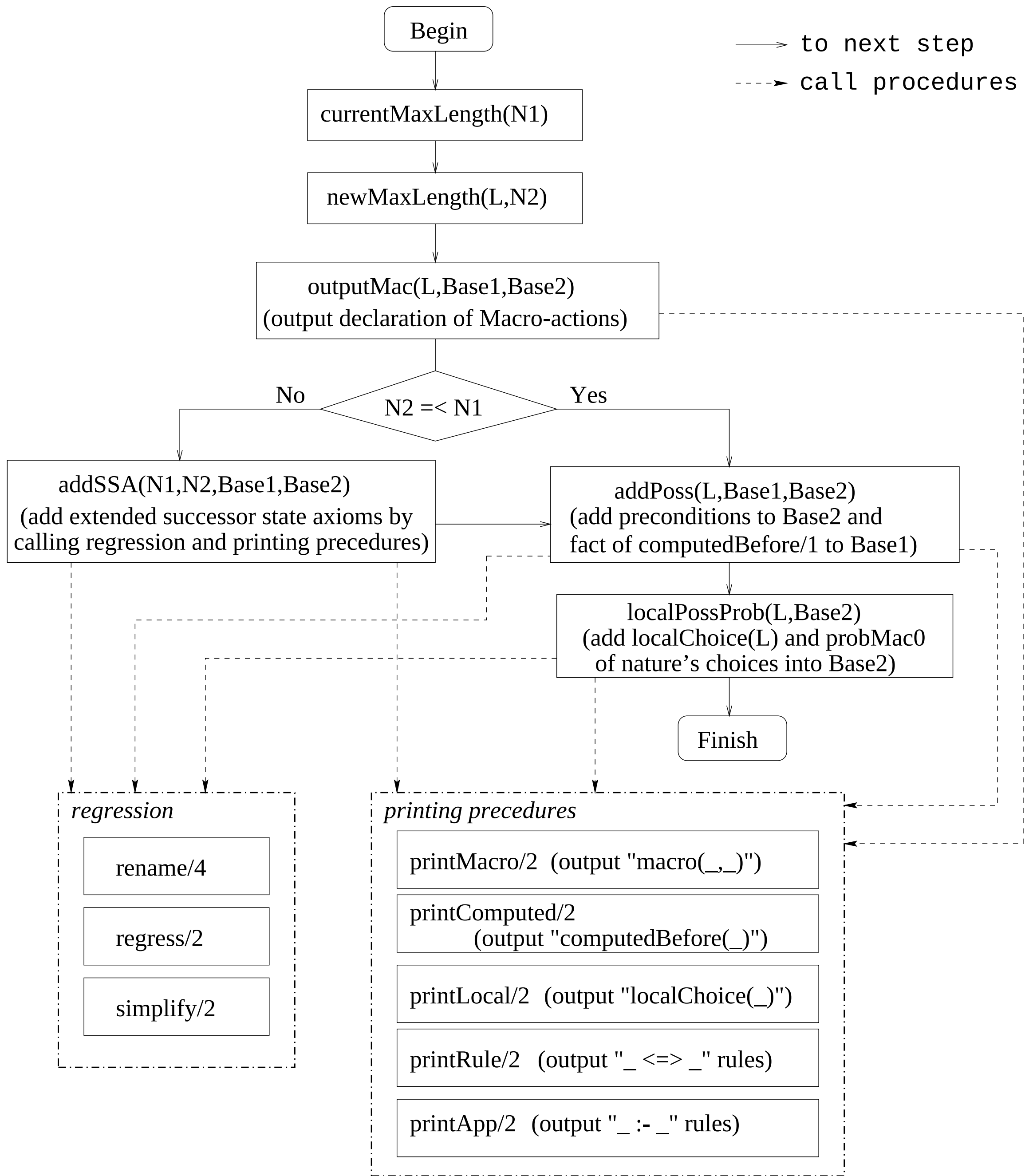

Figure 4.2: The Flow Chart of the Implementation of kbDeveloper(L,Base1,Base2)

and the other knowledge base for later application (as *base2* above) is named *climbBase2* (initially, these two files might be empty, or even don't exist). By executing programs as follows,

---

Example of Developing the Knowledge Base(Static Part)

```
[eclipse 1]: [developer,baseClimb].

developer  compiled traceable 65668 bytes in 0.00 seconds

baseClimb  compiled traceable 11400 bytes in 0.00 seconds




yes.

[eclipse 2]: kbDeveloper([[stepMain(H),liftUpperLeg(H):forwLowLeg:

stepDown(main):moveBarycenter(main):straightLeg],[stepSupp,

forwSupLeg:stepDown(supporting):moveBarycenter(supporting)]],

climbBase1,climbBase2).

/cs/ai/eclipse/lib/lists.pl compiled traceable 8200 bytes in 0.01 seconds

/cs/ai/eclipse/lib/sorts.pl compiled traceable 5420 bytes in 0.01 seconds

/cs/ai/eclipse/lib/strings.pl compiled traceable 6024 bytes in 0.01 seconds


H = H     More? (;)


no (more) solution.
```

---

we developed the knowledge base (static part). Opening *climbBase1* and *climbBase2*, we observe that they include the exact clauses we expected. For example, we have

```
overNewStair(_113, do([_144, _143], _1881)) <=> _143 = forwLowLegS
& _113 = main v _143 = forwSupLegS & _113 = supporting v
(_144 = forwLowLegS & _113 = main v _144 = forwSupLegS &
_113 = supporting v overNewStair(_113, _1881) & -(_144 =
stepDownS(_113))) & -(_143 = stepDownS(_113)).
```

in *climbBase1* which is one of the extended successor state axioms we expected theoretically.

Simplification of the formulas is not easy to deal with during implementation. Because, for example, theoretically we can simplify any clause $C_1 \vee C_2 \vee \cdots \vee C_n$ to be *true* whenever $C_i = p$ and $C_j = \neg p$ for some $i, j \in \{1, 2, \cdots, n\}$ and atom p. During implementation, it will consume too much computing time if we want to do such a thorough simplification provided that the original formulas are not formal, which is not worthy. Therefore, on one hand, we try our best to simply the formula as much as possible; on the other hand, we do not want to consume too much time. Hence, we only did partial simplification, the detailed rules can be found in Appendix C(in regression part).

Chapter 5

The Reuse of the Macro-actions

After having the knowledge base(static part) for macro-actions, we are now interested in the applications: how we introduce macro-actions into high-level programs, how we keep dynamic part of the knowledge base and how we reuse the existing knowledge. Moreover, we will observe the benefit as well as limitation of using macro-actions.

5.1 An Interpreter over Macro-actions: macGolog

As with stGolog, one important use of specified probabilistic domain is in determining how probable some state of affairs will be after an agent performs a stGolog-like program – *macGolog program.* The *macGolog programs* are constructed from stochastic actions and macro-actions together with the Golog program constructors sequence, tests, while loops, conditionals and procedures.

5.1.1 Extending stGolog with Macro-actions

Similar to the stGolog interpreter, we want to specify an interpreter for sequence of combinations of stochastic actions and macro-actions without changing the function of the stGolog interpreter. Our new interpreter, *macDo*, expects a sequence $\alpha_1; \alpha_2; \cdots; \alpha_n; nil$, where every α_i is a stochastic action or a macro-action with body Δ_i, and *nil* is a dummy symbol indicating the end of the sequence. The reason that *nil* is needed is same as the one for *stDo*. We define this interpreter as follows:

$$macDo(nil, p, s, s') \stackrel{def}{=} s = s' \wedge p = 1.$$

Associate the sequence operator to the right:

$$macDo((\alpha; \beta); \gamma, p, s, s') \stackrel{def}{=} macDo(\alpha; (\beta; \gamma), p, s, s').$$

Whenever α is a stochastic action, the definition of *macDo* is same as *stDo*:

$$\begin{aligned} &macDo(\alpha; \beta, p, s, s') \stackrel{def}{=} \\ &\quad \neg(\exists a)[choice(\alpha, a) \wedge Poss(a, s)] \wedge s = s' \wedge p = 1 \vee \\ &\quad (\exists a).choice(\alpha, a) \wedge Poss(a, s) \wedge \\ &\quad (\exists p').macDo(\beta, p', do(a, s), s') \wedge p = prob_0(a, \alpha, s) * p'. \end{aligned} \tag{5.1}$$

Notice that we use $prob_0(a, \alpha, s)$ instead of $prob(a, \alpha, s)$ in the stGolog interpreter, which gives us the same result, since the definition of $prob(a, \alpha, s)$ is

$$prob(a, \alpha, s) = p \equiv Poss(a, s) \wedge p = prob_0(a, \alpha, s) \vee \neg Poss(a, s) \wedge p = 0,$$

hence, we have

$$Poss(a, s) \wedge p = prob(a, \alpha, s) * p' \equiv Poss(a, s) \wedge p = prob_0(a, \alpha, s) * p'. \tag{5.2}$$

Now consider that α is a macro-action and we have had developed the static part's information for it in the knowledge, then

$$
\begin{aligned}
& macDo(\alpha;\beta,p,s,s') \stackrel{def}{=} \\
& \quad \neg(\exists a)[choice(\alpha[1],a) \wedge Poss(a,s)] \wedge s' = s \wedge p = 1 \vee \\
& \quad (\exists c).c \in maxPoss(\alpha,s) \wedge (\exists p_1).p_1 = probMac_0(c,\alpha,s) \wedge \\
& \quad [shorter(c,\alpha) \wedge p = p1 \wedge s' = do(c,s) \vee \\
& \quad \neg shorter(c,\alpha) \wedge macDo(\beta,p_2,do(c,s),s') \wedge p = p1 * p2].
\end{aligned} \tag{5.3}
$$

where $shorter(a,b) \equiv seqLength(a) < seqLength(b)$ for some deterministic sequential action a and some macro-action(either name or body) b. Moreover, up till now, although we gave the definition of $maxPoss(\alpha,s)$, we still did not discuss how to generate it and keep it as fact $maxPossBase(L,\alpha,s)$ practically. The following describes how we develop and retrieve the dynamic part of the knowledge base $maxPoss(\alpha,s)$ for macro-action α in some situation s:

$c \in maxPoss(\alpha,s) \stackrel{def}{=}$

if there exists fact $maxPossBase(L,\alpha,s)$ in the knowledge base for some L,

then $c \in L$ (i.e., element c can be retrieved from base $maxPossBase$);

else call the fact $localChoice(\alpha,V)$ for some list V, compute list $maxposs_0(V,s)$,

assert the fact $maxPossBase(maxposs_0(V,s),\alpha,s)$ and $c \in maxposs_0(V,s)$, (5.4)

where V is a list of possible nature's choices of α according to the definition of $localChoice$, and $maxposs_0(V,s)$ is a list obtained as *for any element* a,

$$
\begin{aligned}
& a \in maxposs_0(V,s) \;\; iff \\
& \quad a \in V \wedge Poss(a,s) \wedge \neg(\exists c)[c \in V \wedge Poss(c,s) \wedge realPrefix(a,c)]
\end{aligned} \tag{5.5}
$$

in which $realPrefix(a,c)$ is true iff a is a prefix of c and $a \neq c$. It is easy to see that the definition of $maxPoss(\alpha,s)$ has the same content as the original definition in Chapter 3.

Lemma 5.1 *The following are satisfied according to our definition of macDo:*

(5-1) *For any situation s,*

$$macDo(\alpha; nil, p, s, s') \equiv stDo(\alpha; nil, p, s, s')$$

if α is a sequence of stochastic actions; and,

(5-2) *for any situation s, let α be a sequence of stochastic actions, β be a macro-action with the body Δ (including the special case that there is no actions before β, i.e., sequence α doesn't exist), and γ be a sequence of combinations of stochastic actions and macro-actions followed by nil (including the special case that $\gamma = nil$), we have*

$$macDo(\alpha; \beta; \gamma, p, s, s') \equiv macDo(\alpha; \Delta; \gamma, p, s, s').$$

Proof: (1) To property (5-1), it is directly from the definition of *macDo*(5.1) and the result(5.2) when α is stochastic action.

(2) The proof of property (5-2) for the general case that α is a finite sequence of stochastic actions is similar to the case that α is a stochastic action, therefore, we present the proof for α is a stochastic action only.

If there is no primitive action a satisfying that $choice(\alpha, a) \wedge Poss(a, s)$, then

$$macDo(\alpha; \beta; \gamma, p, s, s') \equiv p = 1 \wedge s = s' \equiv macDo(\alpha; \Delta; \gamma, p, s, s');$$

otherwise, to prove

$$macDo(\alpha; \beta; \gamma, p, s, s') \equiv macDo(\alpha; \Delta; \gamma, p, s, s')$$

is equivalent to prove for every primitive action a satisfying that $choice(\alpha, a) \wedge Poss(a, s)$,

$$macDo(\beta; \gamma, p', do(a, s), s') \equiv macDo(\Delta; \gamma, p', do(a, s), s').$$

There are three cases for any deterministic sequential action c:

(a) If $\neg(\exists c)[choice(\beta[1], c) \wedge Poss(c, do(a, s))]$, then

$macDo(\beta; \gamma, p', do(a, s), s') \equiv s' = do(a, s) \wedge p' = 1 \equiv macDo(\Delta; \gamma, p', do(a, s), s')$.

(b) If $(\exists c).c \in maxPoss(\beta, do(a, s)) \wedge \neg shorter(c, \beta)$, then according to the definition of $macPoss$, we have $Poss(c, do(a, s)) \wedge (\wedge_{i=1}^{m} choice(\Delta[i], c[i]))$ where $(m = seqLength(\Delta))$ and therefore, for such c,

$$
\begin{aligned}
& macDo(\beta; \gamma, p', do(a, s), s') \\
& \equiv (\exists p_1).p_1 = probMac_0(c, \beta, do(a, s)) \wedge \\
& \qquad macDo(\gamma, p2, do(c, do(a, s)), s') \wedge p' = p_1 * p_2 \\
& \equiv (\exists p_1).p_1 = prob_0(c[1], \beta[1], do(a, s)) * \cdots * prob_0(c[m], \beta[m], \\
& \qquad do(c[1]; \cdots ; c[m-1], do(a, s)) \wedge macDo(\gamma, p2, do(c, do(a, s)), s') \wedge p' = p_1 * p_2 \\
& \equiv macDo(\Delta; \gamma, p', do(a, s), s')
\end{aligned}
$$

according to $do(A_1; A_2; \cdots ; A_n, s) = do(A_n, \cdots , do(A_1, s) \cdots)$ and the definition of $macDo$.

(c) If $(\exists c).c \in maxPoss(\beta, do(a, s)) \wedge shorter(c, \beta)$, for such c, let $t = seqLength(c)$, and we have $Poss(c, do(a, s)) \wedge (\wedge_{i=1}^{t} choice(\Delta[i], c[i]))$ and $\neg(\exists d)[choice(\Delta[t + 1], d) \wedge Poss(d, do(c, do(a, s)))]$, therefore, for such c,

$$
\begin{aligned}
& macDo(\beta; \gamma, p', do(a, s), s') \\
& \equiv p' = probMac_0(c, \beta, do(a, s)) \wedge s' = do(c, do(a, s)) \\
& \equiv p' = prob_0(c[1], \beta[1], do(a, s)) * \cdots * prob_0(c[m], \beta[m], \\
& \qquad do(c[1]; \cdots ; c[m-1], do(a, s)) \wedge s' = do(c, do(a, s)) \\
& \equiv macDo(\Delta[1]; \cdots ; \Delta[t]; \Delta[t+1] : nil, p', do(a, s), s') \\
& \equiv macDo(\Delta; \gamma, p', do(a, s), s')
\end{aligned}
$$

according to $do(A_1; A_2; \cdots ; A_n, s) = do(A_n, \cdots , do(A_1, s) \cdots)$ and the definition of $macDo$.

Hence, we proved property (5-2). ■

We therefore can get following properties:

Theorem 5.2 *For any situation s and any sequence $\alpha_1; \alpha_2; \cdots; \alpha_n$ where every α_i is either stochastic action or macro-action with body Δ_i, we have*

$$macDo(\alpha_1; \alpha_2; \cdots; \alpha_n; nil, p, s, s') \equiv stDo(\beta_1; \beta_2; \cdots; \beta_n; nil, p, s, s'),$$

where every β_i either is α_i if α_i is a stochastic action, or is Δ_i if α_i is a macro-action.

Proof: According to property (5-2), we have

$$macDo(\alpha_1; \alpha_2; \cdots; \alpha_n; nil, p, s, s') \equiv macDo(\beta_1; \beta_2; \cdots; \beta_n; nil, p, s, s'),$$

by replacing the macro-actions with their bodies from left to right and according to property (5-1), we have

$$macDo(\beta_1; \beta_2; \cdots; \beta_n; nil, p, s, s') \equiv stDo(\beta_1; \beta_2; \cdots; \beta_n; nil, p, s, s'),$$

therefore, our theorem is proved. ■

This property indicates that although we extend the interpreter with macro-actions, we didn't change the function of the stGolog interpreter. So what's the advantage for using knowledge base? It is for the purpose of saving computational time, which will be discussed later.

5.1.2 Generating the Dynamic Part of the Knowledge Base

Our purpose of developing the dynamic part of the knowledge base is to keep necessary sets of $maxPoss$ for macro-actions in some particular situations for the sake of reuse. Imagining the example of robot climbing stairs with macro-actions $stepMain(h)$ and $stepSupp$, we are interested in the robot climbing (legal) stairs repeatedly from the local initial situations which are same as S_0, and suppose we keep the set of maximal possible choices $maxPoss(stepMain(15), S_0)$ and the set $maxPoss(stepSupp, do([liftTill(15), forwLowLegS, stepDownS(main), moveBarycenterS(main), straightLeg], S_0))$. When the controller calls procedure(3.5) *climbing*(15) again, the robot can directly retrieve these information without re-computation. During the first time of calling procedure(3.5) *climbing*(15), we can use command $assert/1$ to help us achieve above description of keeping information of $maxPoss/2$ for macro-actions on object instance in certain situations as facts $maxPossBase/3$. Similarly, if we have macro-action(3.6) $climbStair(h)$ instead of macro-actions $stepMain(h)$ and *stepSupp*, we will save $maxPoss(climbStair(15), S_0)$.

But, only using $assert/1$ command keeps very narrow knowledge, i.e. only for macro-actions on particular object instance and situation instances. Thinking of stairs of height 15 and 17, if we perform macro-action $stepMain(15)$ and $stepMain(17)$ respectively in the local initial situation S_0, we will get the same set of $maxPoss$ regardless the difference of objects 15 and 17. Therefore, we are considering extend $maxPossBase(L, \alpha(\vec{x}), S)$ for situation instance S and macro-action $\alpha(\vec{x})$ with variable parameters $\vec{x}$, so that $maxPossBase(L, \alpha(\vec{x}), S)$ represents a uniform fact for certain class of objects.

Without loss of generality, any system which can described in the situation calculus can have an equivalent description in the situation calculus satisfying the following condition: for every atomic sentence, definition of procedure or definition of macro-action, if it has augments that are same as the arguments of some primitive action function, then these augments have the same relative order both in the atomic sentence and in the action function. For example, suppose that in a system we have atomic sentence $F(y_1, y_2, y_3, s)$ and action functions $a(x_1, x_2)$ and $b(x)$ and that according to the basic action theory we know that y_1 (respectively, y_3, y_2) represents the same object with augment x_1 (respectively, x_2, x), then the relative order of y_1 and y_3 (respectively, of y_2) are same as the order of x_1 and x_2 (respectively, of x). Given any system $\mathcal{D}$ satisfying above condition, we give the following definition of *ob-class* for $\mathcal{D}$.

Definition 5.3 *Given a system $\mathcal{D} = \mathcal{D}_{S_0} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss}$, suppose that the set of objects in $\mathcal{D}$ is L and the set of objects appearing in the $\mathcal{D}_{ap} \cup \mathcal{D}_{ss}$ is L_1,*

(1) *every object O in L_1 is a* 1-ary ob-class,

(2) *for any objects O_1 and O_2 in set $L \backslash L_1$ (can be the same object), we say that O_1 and O_2 are in the same* 1-ary ob-class *iff for every relational sentence $F(x_1, x_2, \cdots, x_m)$ (including fluent and non-fluent) in $\mathcal{D}_{S_0}$ where $m \geq 1$, we interpret any augment x_i $(i = 1, 2, \cdots, m)$ with O_1 and O_2 respectively, it is true that either both of O_1 and ${}_2$ do not belong to the domain of x_i, or the two interpreted sentences return the same truth value (i.e., both of the sentences $F(x_1, x_2, \cdots, O_1, \cdots, x_m)$ and $F(x_1, x_2, \cdots, O_2, \cdots, x_m)$ are either unsatisfiable, satisfiable, or tautological).*

*For any n-ary $(n > 1)$ object vectors $O_1 = (O_1^1, \cdots, O_1^n)$, $O_2 = (O_2^1, \cdots, O_2^n) \in L^n$, we say that O_1 and O_2 are in the same n-*ary ob-class *iff the following two conditions are satisfied:*

(I) *for every i $(1 \leq i \leq n)$, O_1^i and O_2^i are in the same 1-ary ob-class and*

(II) *for every relational sentence $F(x_1, x_2, \cdots, x_m)$ (including fluent and non-fluent) in $\mathcal{D}_{S_0}$ where $m \geq n$, we interpret any n ordered augments $x_{i_1}, x_{i_2}, \cdots, x_{i_n}$ $(i_1 < i_2 < \cdots < i_n)$ in F with O_1 and O_2 respectively, it is true that either both of O_1 and O_2 do not belong to the domain of this sequence of augments, or the two interpreted sentences return the same truth value (i.e., both are either unsatisfiable, satisfiable, or tautological).*

Suppose we defined an macro-action $\alpha(y_1, y_2, \cdots, y_n)$ with body $a_1(y_{(1,1)}, \cdots, y_{(1,1_t)}); \cdots; a_m(y_{(m,1)}, \cdots, y_{(m,m_t)})$ where every (i, j) $(1 \leq i \leq m,\ 1 \leq j \leq i_t)$ is a natural number in $[1..n]$ and $(i, 1) < (i, 2) < \cdots < (i, i_t)$ for every i, we say object vectors $O_1 = (O_1^1, O_1^2, \cdots, O_1^n)$, $O_2 = (O_2^1, O_2^2, \cdots, O_2^n) \in L^n$ are in the same ob-class *for macro-action α iff O_1 and O_2 are both in the domain of $(y_1, y_2, \cdots, y_n)$, and $(O_1^{(i,1)}, O_1^{(i,2)}, \cdots, O_1^{(i,i_t)})$ and $(O_2^{(i,1)}, O_2^{(i,2)}, \cdots, O_2^{(i,i_t)})$ are in the same i_t-ary ob-class for every i.*

For instance, in the example of robot climbing stairs, we have

$$L = \{wrongPos, main, supporting\} \cup \{r | r \text{ is a non-negative real number}\},$$

and
$$L1 = \{wrongPos, main, supporting\},$$

the set of 1-ary ob-classes is

$\{\{wrongPos\}, \{main\}, \{supporting\}, \{r|r \in \mathcal{R} \wedge 0 < r < 20\}, \{r|r \in \mathcal{R} \wedge r \geq 20\}, \{0\}\}$. And as an example, all the numbers greater than 0 and less than 20 are in the same ob-class for macro-action $stepMain(h)$.

Lemma 5.4 *Given a system $\mathcal{D}$ described in the situation calculus, suppose object vectors O_1 and O_2 are in the same n-ary ob-class, then for any S_0-regressable relational formula $F(x_1, x_2, \cdots, x_n)$ in $\mathcal{L}'_{sc}$, either both O_1 and O_2 are not in the domain of F's arguments, or both of them are in the domain and if we interpret the variables with O_1 and O_2 respectively, the formula returns the same truth value.*

Proof: We can prove it by induction on the longest number m of primitive actions for the situations from S_0 in F.

Base Case: $m \leq 1$, i.e., S_0 is the only situation in F(if any), then according to the definition of n-ary ob-class, it is true for all sentences in $\mathcal{D}_{S_0}$. Moreover, if F is of $Poss$ atom, $\mathcal{R}^*[F]$ is a formula composed of fluent or non-fluent atoms with first order connectives, therefore, it is easy to prove the proposition is true using induction on the length of $\mathcal{R}^*[F]$, again since $F \equiv \mathcal{R}^*[F]$, it is also true for F. Therefore, for general F, according to $F \equiv \mathcal{R}^*[F]$ and the definition of the syntactic form of the formulas in $\mathcal{L}'_{sc}$, it is easy to see the proposition is true.

Induction Step: Suppose this proposition is true for all $1 \leq j < m$, now we are going to prove it for m, notice that $\mathcal{R}^*[F(x_1, x_2, \cdots, x_n)] = \psi_F(x_1, x_2, \cdots, x_n)$ is a formula uniform in S_0, therefore according to the hypothesis and $F \equiv \mathcal{R}^*[F]$ the proposition is true for m.

Hence, we proved the proposition for all $m \in \mathcal{N}$. ■

Therefore, according to above Lemma and the definition of ob-class, we can get the following theorem:

Theorem 5.5 *Given a system $\mathcal{D}$ described in the situation calculus, suppose we have* *a macro-action* $\alpha(y_1, y_2, \cdots, y_n)$ *and situation instance* S *beginning from* S_0*, i.e.* $S = do([A_1, \cdots, A_n], S_0)$*, then for any object vectors* O_1 *and* O_2 *which are in the same* ob-class *for* α*, we have*

$$maxPoss(\alpha(O_1), S) = maxPoss(\alpha(O_2), S).$$

Therefore, we can extend the *maxPossBase* of macro-actions in any situation instance from particular objects to ob-classes by renaming the objects in the macro-action with new variables, so that reduce the scales of the dynamic part of the knowledge base. We hereby demand that it is the controller's responsibility to provide objects in the same ob-class for macro-actions. If we need to work on objects in different ob-classes, we need to erase the former dynamic part of the knowledge base. We certainly can somehow retract the rules, but right now, to make our lifer easier, we only need to quit the running system and reload it again, since we use *assert*/1 command to keep the information of *maxPoss*. The practical implementation of the interpreter can be seen the next subsection.

5.1.3 The macGolog Interpreter

Other descriptions of *macDo* are same as *stDo* on other Golog constructors, therefore the full *macGolog* interpreter is as follows:

An macGolog (Macro-action Golog) Interpreter

```
macDo(nil,1,S,S):- !.
macDo(A : B,P,S1,S2) :- stochastic(A), !, % A is a stochastic action
   (not (choice(A,C), poss(C,S1)), !,     % Program can't continue.
    S2 = S1, P = 1 ;                      % Create a leaf.
```

```
  % once is an Eclipse Prolog built-in. once(G) succeeds the first time
  % G succeeds, and never tries again under backtracking. We use it here
  % to prevent macDo from generating the same leaf situation more than
  % once, when poss has multiple solutions.
    choice(A,C), once(poss(C,S1)), prob0(C,A,S1,P1),
    macDo(B,P2,do([C],S1),S2), P is P1 * P2 ).
macDo(A : B,P,S1,S2) :- macro(A,A1:A2), !,  % A is a macro-action
    (not (choice(A1,C), poss(C,S1)), !,     % Program can't continue.
     S2 = S1, P = 1 ;                       % Create a leaf.
     maxposs(C,A,S1), probMac0(C,A,S1,P1),
     (shorter(C,A1:A2),
      P is P1, S2=do(C,S1);               % Program can't continue.
      not shorter(C,A1:A2),
      macDo(B,P2,do(C,S1),S2), P is P1 * P2)).
macDo((A : B) : C,P,S1,S2) :- macDo(A : (B : C),P,S1,S2).
macDo(?(T) : A,P,S1,S2) :- holds(T,S1), !, macDo(A,P,S1,S2) ;
                           S2 = S1, P = 1. % Program can't continue.
                                           % Create a leaf.
macDo(if(T,A,B) : C,P,S1,S2) :- holds(T,S1), !, macDo(A : C,P,S1,S2) ;
                                macDo(B : C,P,S1,S2).
macDo(A : B,P,S1,S2) :- proc(A,C), macDo(C : B,P,S1,S2).
macDo(while(T,A) : B,P,S1,S2) :- holds(T,S1), !,
                                 macDo(A : while(T,A) : B,P,S1,S2) ;
                                 macDo(B,P,S1,S2).

% shorter(C,A): list C is shorter than macro-action A
shorter([_],_:_):- !.
shorter([_|T], _:A2):- shorter(T,A2).
```

```
% maxposs(C,A,S): C is an element of maxPoss(A,S)
maxposs(C,A,S):-
   maxPossBase(List,A,S),!,     % If we've computed local optimal results
   member(C,List);              % before, retrieve it from database
   not maxPossBase(List,A,S),!, % otherwise, computed "maxposs" according
   localChoice(A,V),            % to the definition, and save the solution
   maxposs0(List,V,S),          % by asserting new rule "maxPossBase" to
   generalize(A,List,A1,List1), % generate the knowledge base(dynamic part).
   asserta(maxPossBase(List1,A1,S)), !,
   member(C,List).

% maxposs0(L,A,S): L is the set of maxPoss(A,S)
maxposs0([],[],_).
maxposs0([C|T1],[C|T],S):- once(poss(C,S)), !,
                           removeSub(C,T,L1), maxposs0(T1,L1,S).
maxposs0(T1,[C|T],S):- not poss(C,S), !, maxposs0(T1,T,S).

% removeSub(C,T,T1):remove every prefix of C in T.
removeSub(_,[],[]):- !.
removeSub(C,[A|T],[A|T1]):- not append(A,_,C), !, removeSub(C,T,T1).
removeSub(C,[A|T],T1):- append(A,_,C), !, removeSub(C,T,T1).

% generalize(A,List,A1,List1): replace objects in A and List with variables
% to get A and List, hence generalize the usage of knowledge to ob-class
generalize(A,List,A1,List1):-
   A =.. [_|P], generalize0(P,A,List,A1,List1).

generalize0([],A,List,A,List):- !.
generalize0([H|T],A,List,A1,List1):-
```

```
    generalize1(H,_,A,List,A0,List0),
    generalize0(T,A0,List0,A1,List1).

generalize1(H,G,A,List,A0,List0):-
       sub(H,G,A,A0), sub_list(H,G,List,List0).

poss([A],S):- poss(A,S), !.
stochastic(A) :- choice(A,N), !.

maxPossBase([],nil,_). % special database to avoid no def. of "maxPossBase"
macro(nil,nil).        % special database to avoid no def. of macro

% 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).

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)).
```

According to Theorem 5.2, Theorem 5.5, the announcement that the controller won't operate on different ob-class objects under the same dynamic knowledge base, and that we have the same description of *macDo* for the constructors tests, while loops, conditionals and procedures as of *stDo*, by using induction proof, we can easily get

Theorem 5.6 *For any situation s and any macGolog program α,*

$$macDo(\alpha; nil, p, s, s') \equiv stDo(\beta; nil, p, s, s')$$

for some number p and situation s', where β is a stGolog program obtained by replacing every macro-action in α with its body.

This is exactly what we mean not changing the function of the stGolog interpreter. We also can define the probabilities that some situation-suppressed sentence ψ will be true after executing a macGolog program γ:

$$probF(\psi, \gamma) \stackrel{def}{=} \sum_{\{(p,\sigma) \mid \mathcal{D} \models macDo(\gamma; nil, p, S_0, \sigma) \wedge \psi[\sigma]\}} p. \tag{5.6}$$

Here, $\mathcal{D}$ stands for the background basic action theory. Notice that this definition is same as the definition given in the stGolog. The implementation is straightforward and we name the following file as *macProbF*.

Probabilities for macGolog Programs

```
probF(F,Prog,Prob) :- findall(P,
                              S^(macDo(Prog : nil,P,s0,S), once(holds(F,S))),
                              PS),
                      addNumbers(PS,Prob).
addNumbers([],0.0).
addNumbers([N | Ns],Sum) :- addNumbers(Ns,Sum1), Sum is Sum1 + N.
```

5.2 The Experiments and Discussion

5.2.1 The Experiment of the correctness

We continue to consider the example of robot climbing stairs, and suppose we have computed the knowledge base (static part) as *climbBase*2 given in Chapter 4.3, the macGolog interpreter is saved as file *macGolog*, the complete specification of robot climbing stairs in Prolog clauses is saved as file *climb* (cf. Appendix D and notice its different forms from file *baseClimb* in previous chapter). We now compute some probabilities after loading all these files. Since we would like the extended axioms to have higher priorities, we will load *climbBase*2 before *climb*.

Computing Probabilities for macGolog Programs

```
[eclipse 1]: [macGolog,climbBase2,climb,macProbF].
```

```
macGolg    compiled optimized 17132 bytes in 0.00 seconds
climbBase2 compiled optimized 99508 bytes in 0.03 seconds
climb      compiled optimized 11496 bytes in 0.00 seconds
macProbF   compiled optimized 804 bytes in 0.00 seconds

yes.
[eclipse 2]: macDo(?(legalStair(15)):stepMain(15):stepSupp:nil,P,s0,S).
/cs/ai/eclipse/lib/lists.pl compiled optimized 7628 bytes in 0.00 seconds

P = 0.130434781
S = do([malfunc(15)], s0)     More? (;)

P = 0.303685099
S = do([forwSupLegS, stepDownS(supporting), moveBarycenterS(supporting)],
    do([liftTill(15), forwLowLegS, stepDownS(main), moveBarycenterS(main),
    straightLeg], s0))     More? (;)

P = 0.0759212747
S = do([forwSupLegS, stepDownS(supporting), moveBarycenterF(supporting)],
    do([liftTill(15), forwLowLegS, stepDownS(main), moveBarycenterS(main),
    straightLeg], s0))     More? (;)

P = 0.0421784893
S = do([forwSupLegS, stepDownF(supporting)], do([liftTill(15), forwLowLegS,
    stepDownS(main),  moveBarycenterS(main), straightLeg], s0))     More? (;)

P = 0.105446219
S = do([forwSupLegF], do([liftTill(15), forwLowLegS, stepDownS(main),
    moveBarycenterS(main), straightLeg], s0))     More? (;)
```

```
P = 0.131807774

S = do([liftTill(15), forwLowLegS, stepDownS(main), moveBarycenterF(main)],
    s0)    More? (;)

P = 0.0732265413

S = do([liftTill(15), forwLowLegS, stepDownF(main)], s0)     More? (;)

P = 0.137299761

S = do([liftTill(15), forwLowLegF], s0)     More? (;)

no (more) solution.
[eclipse 3]: probF(true,stepMain(15):stepSupp,P).

P = 0.999999881     More? (;)

no (more) solution.
[eclipse 4]: macDo(stepMain(17):forwSupLeg:nil,P,s0,S).

P = 0.145299152

S = do([malfunc(17)], s0)     More? (;)

P = 0.406027

S = do([forwSupLegS], do([liftTill(17), forwLowLegS, stepDownS(main),
    moveBarycenterS(main), straightLeg], s0))     More? (;)

P = 0.101506747

S = do([forwSupLegF], do([liftTill(17), forwLowLegS, stepDownS(main),
    moveBarycenterS(main), straightLeg], s0))     More? (;)
```

```
P = 0.126883432
S = do([liftTill(17), forwLowLegS, stepDownS(main),
    moveBarycenterF(main)], s0)     More? (;)

P = 0.0704907924
S = do([liftTill(17), forwLowLegS, stepDownF(main)], s0)     More? (;)

P = 0.149792925
S = do([liftTill(17), forwLowLegF], s0)     More? (;)

no (more) solution.
[eclipse 5]: probF(overNewStair(main),stepMain(17):forwSupLeg,P).

P = 0.0704907924     More? (;)

no (more) solution.
```

We also ran these examples under stGolog interpreter by replacing the macro-actions with their bodies and viewing them as stGolog programs. We get the same probabilities in the same situations. As to the example, we can see that our macGolog interpreter works well and has the same function as the stGolog interpreter. But what is the benefit of using the macGolog interpreter over macro-actions? As we discussed before, we are focusing on the class of problems that we expect the agent to perform the same (or even part of the same) strategies or programs repeatedly when it is in the same local environment as we discussed in Chapter 3.1. For instance, in above robot climbing stairs example, if we perform the *climbing*(h) procedure repeatedly for legal stairs of height h provided that the controller can reset the robot's status to be same as initial situation (we call it local initial situation) when malfunction occurs after performing the *climbing(h)* procedure or that there is no malfunction occurs, i.e., the robot performs the *climbing(h)* procedure successfully, it will "forget" what it performed, (might somehow count the stairs first, which is not necessary), and reset its own situation to be the local initial situation. If we use the stGolog interpreter to compute the probabilities of the outcomes every time the agent calls the *climbing(h)* procedure in the local initial situation, then one branch of the following computational tree (e.g. Figure 5.1) is always gone through step by step, which is time-consuming.

If we introduce macro-actions $stepMain(h)$ and $stepSupp$ as (3.3) and (3.4) in Chapter 3, then procedure $climbing(h)$ can be represented as (3.5). By using macGolog interpreter to get the probabilities of the deterministic performance every time the agent calls the *climbing(h)* for legal stairs of height h in the local initial situation, except the first time in which we need to compute all the probabilities step by step and the computational tree looks like Figure 5.1. Other times when we recall the procedure in the local situation, the computational tree of probabilistic outcomes looks like Figure 5.2, even can be as shorter as the following tree (c.f Figure 5.3) if the stair of height H(instant number) has been climbed before.

If we consider to define macro-action as $climbStair(h)$(3.6) in chapter 3, the recall of procedure(3.7) $climbing(h)$ for any stairs of legal height would have even shorter outcome tree (only with 2 steps). Therefore, regardless the scales of the knowledge base, the computational time of reusing macro-actions is smaller than not using macro-actions.

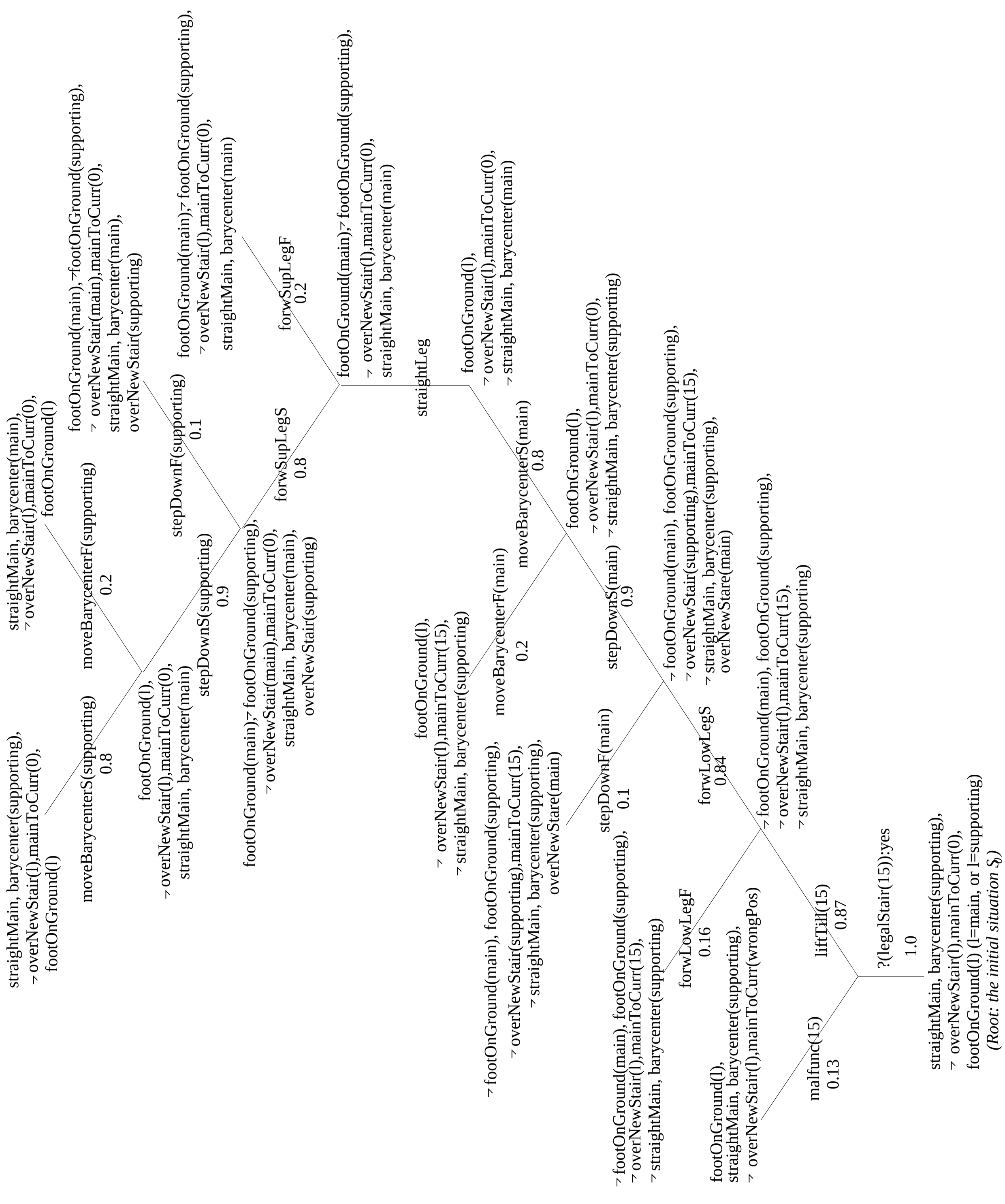

Figure 5.1: The Tree of Possible Outcomes for Running Procedure(3.1) *climbing*(15) under the stGolog Interpreter

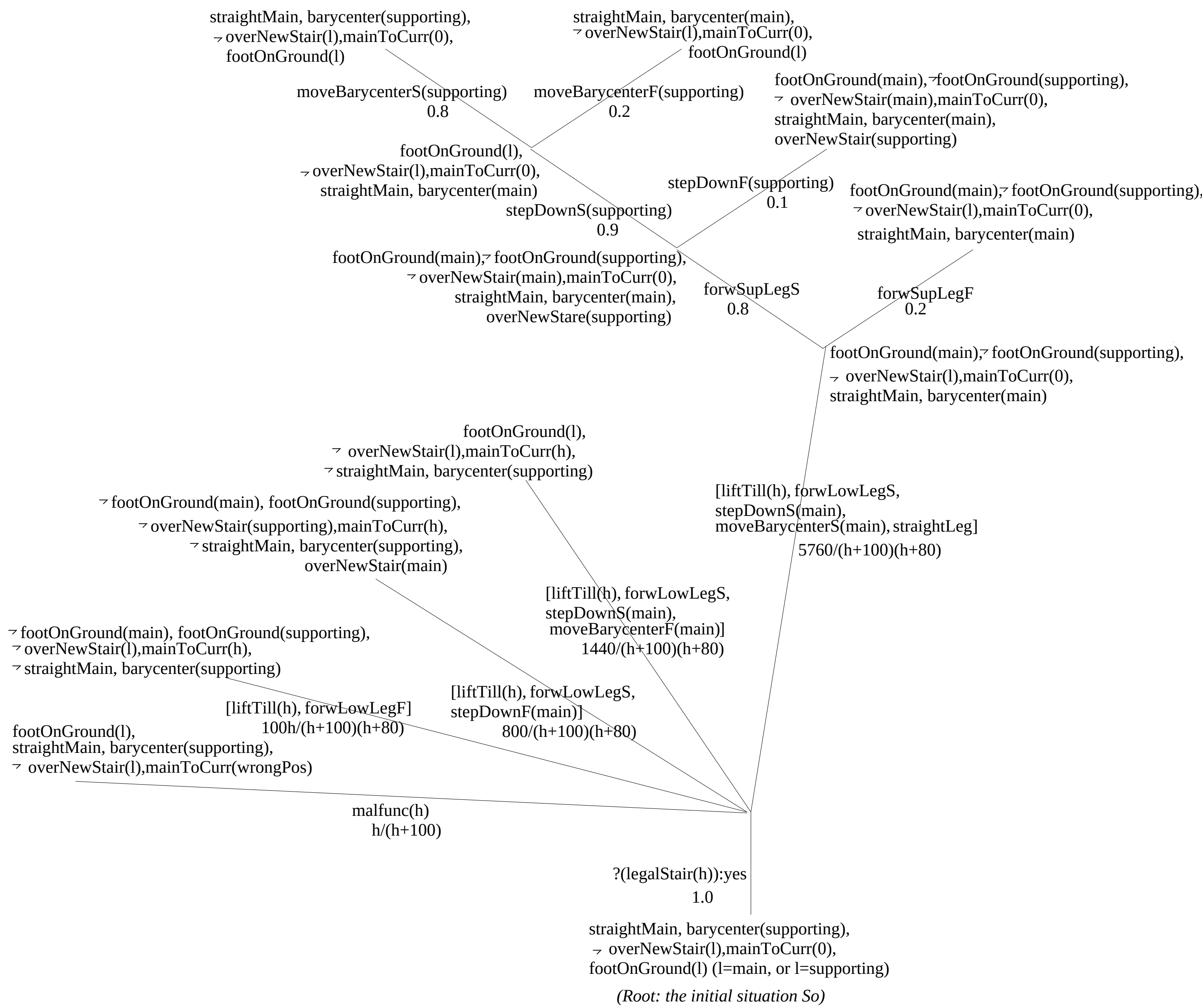

Figure 5.2: The Tree of Possible Outcomes for Repeating Procedure(3.5) *climbing*(h) for Legal height stairs

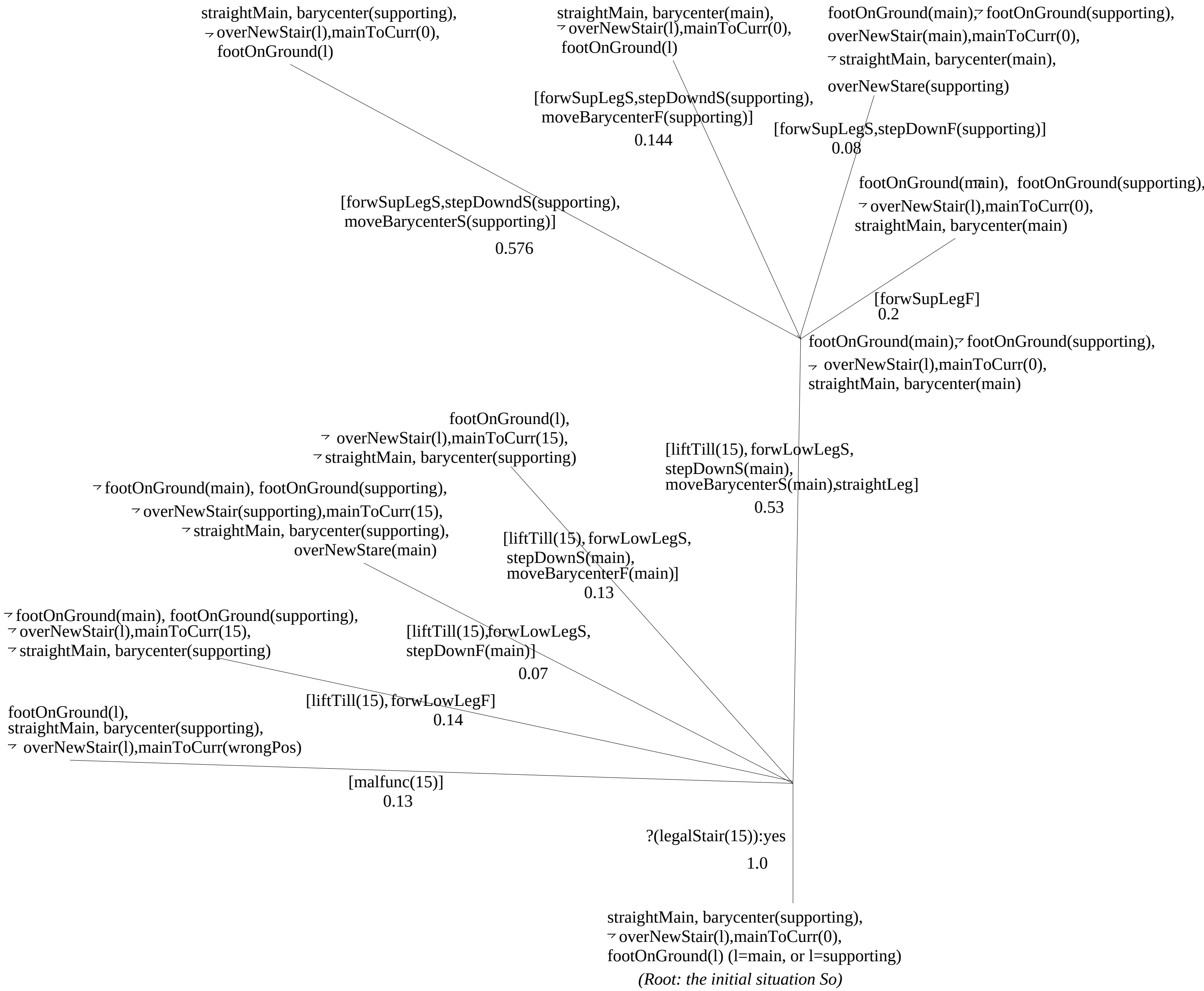

Figure 5.3: The Tree of Possible Outcomes for Repeating Procedure(3.5) *climbing*(15) Provided That the Robot Has Climbed Stair of Height 15 Before

5.2.2 Experiments of Comparison

But, in practice, the scales of the knowledge base will affect the computational time. To make detailed observation and discussion, we did different tests for different cases of whether or not using macro-actions, and how to define macro-actions over the various environments of whether the stairs' heights change frequently. To make our life easier, in the experiments we did not describe the behaviors of the step-in by the controller or the reset actions by the robot, but only abstracted them to be simply resetting the situation to S_0 after calling the climbing procedure every time, because currently we only want to concentrate on checking if the reuse of macro-actions really saves time in a long run.

We denote procedure(3.5) $climbing(h)$ defined with two macro-actions as *Exp.1*, denote procedure(3.7) $climbing(h)$ defined with one macro-action as *Exp.2* supposing that we've developed the static parts of the knowledge bases for *Exp.1* and *Exp.2* respectively, and denote procedure(3.1) $climbing(h)$ running under the stGolog interpreter as *Exp.3*. By the way, comparing with the time of reusing macro-actions, the time of developing static part is fixed and much smaller. For instance, in the example of Robot Climbing Stairs it only takes CPU time 0.01 seconds to obtain $climbBase1$ and $climbBase2$ for macro-actions (3.3) and (3.4), suppose we will reuse these two macro-actions for n times, we have $lim_{n\rightarrow\infty}\frac{0.01}{n} = 0$. Therefore, we will ignore it and concentrate on the experiments of applications of using and reusing macro-actions. Notice that the definition *probF* could gather all the possible outcomes for a program, therefore, we used it to do the tests. For the purpose of simulating the frequency of the change of the stairs' heights, we gave following five tests:

Five Tests in Prolog

```
% test1: All N stairs are of the same height.
test1(N,T):- cputime(T1), test0(N), cputime(T2), T is T2-T1.
```

```
test0(0):-!.

test0(N):- N>0, !, probF(true, climbing(15), _), N1 is N-1, test0(N1).


% test2: There are at most 10 different heights for the N stairs.

test2(N,T):- cputime(T1), test02(N), cputime(T2), T is T2-T1.

test02(0):-!.

test02(N):- N>0, !, mod(N,10, H), H1 is H+10,

            probF(true, climbing(H1), _), N1 is N-1, test02(N1).


% test3: There are at most 50 different heights for the N stairs.

test3(N,T):- cputime(T1), test03(N), cputime(T2), T is T2-T1.

test03(0):-!.

test03(N):- N>0, !, mod(N,50, H), H1 is 5+H/4,

            probF(true, climbing(H1), _), N1 is N-1, test03(N1).


% test4: There are at most 800 different heights for the N stairs.

test4(N,T):- cputime(T1), test04(N), cputime(T2), T is T2-T1.

test04(0):-!.

test04(N):- N>0, !, mod(N,800, H), H1 is 1+H/50,

            probF(true, climbing(H1), _), N1 is N-1, test04(N1).


% test5: The N stairs' heights are random, i.e., they might be all different.

test5(N,T):- cputime(T1), test05(N), cputime(T2), T is T2-T1.

test05(0):-!.

test05(N):- N>0, !, frandom(H), H1 is H*20,

            probF(true, climbing(H1), _), N1 is N-1, test05(N1).
```

Clearly, from test suit *test1* to test suit *test5* the change of the stairs' heights becomes more and more frequently. We run procedure *Exp.1* (*Exp.2* and *Exp.3* respectively) on each of the five test suits for values of N from 100 to 2000 with a step-size of 100 and repeat it five times for every value of N to get the CPU time T for each trial. The CPU times presented in the following Table 5.1 (respectively, Table 5.2 and Table 5.3) are the average over the five distinct trials (to reduce the measurement error) with unit *second*:

N	test1	test2	test3	test4	test5
100	0.050	0.066	0.116	0.184	0.188
200	0.106	0.112	0.168	0.416	0.402
300	0.148	0.162	0.216	0.688	0.678
400	0.198	0.200	0.276	1.024	0.992
500	0.238	0.258	0.318	1.376	1.372
600	0.270	0.296	0.380	1.866	1.810
700	0.326	0.340	0.436	2.354	2.286
800	0.378	0.382	0.464	2.910	2.802
900	0.430	0.424	0.516	3.098	3.404
1000	0.466	0.470	0.572	3.374	4.028
1100	0.514	0.514	0.620	3.586	4.704
1200	0.578	0.560	0.674	3.806	5.496
1300	0.582	0.614	0.718	3.898	6.274
1400	0.626	0.648	0.778	4.078	7.164
1500	0.682	0.716	0.838	4.198	8.144
1600	0.728	0.736	0.904	4.312	9.006
1700	0.742	0.774	0.942	4.532	10.174
1800	0.800	0.848	0.968	4.640	11.104
1900	0.856	0.876	1.050	4.740	12.272
2000	0.912	0.918	1.092	4.882	13.508

Table 5.1: Experiment Results of Computational Time for *Exp.1*

N	test1	test2	test3	test4	test5
100	0.078	0.078	0.084	0.080	0.078
200	0.138	0.136	0.142	0.146	0.146
300	0.200	0.210	0.204	0.210	0.208
400	0.270	0.270	0.272	0.282	0.274
500	0.330	0.344	0.340	0.340	0.338
600	0.390	0.400	0.402	0.406	0.398
700	0.458	0.464	0.470	0.470	0.466
800	0.520	0.528	0.538	0.530	0.536
900	0.588	0.588	0.600	0.600	0.594
1000	0.654	0.656	0.658	0.660	0.664
1100	0.726	0.720	0.726	0.722	0.722
1200	0.774	0.792	0.788	0.794	0.790
1300	0.846	0.848	0.856	0.862	0.854
1400	0.908	0.910	0.932	0.922	0.924
1500	0.968	0.967	0.984	0.984	0.980
1600	1.034	1.038	1.058	1.058	1.048
1700	1.098	1.102	1.116	1.114	1.108
1800	1.164	1.182	1.170	1.182	1.180
1900	1.232	1.228	1.266	1.254	1.250
2000	1.304	1.292	1.340	1.318	1.300

Table 5.2: Experiment Results of Computational Time for *Exp.2*

N	test1	test2	test3	test4	test5
100	0.105	0.092	0.090	0.094	0.094
200	0.202	0.176	0.182	0.178	0.182
300	0.280	0.266	0.268	0.266	0.268
400	0.378	0.342	0.356	0.350	0.360
500	0.468	0.440	0.438	0.434	0.450
600	0.550	0.520	0.530	0.520	0.532
700	0.638	0.618	0.616	0.612	0.624
800	0.754	0.694	0.700	0.688	0.698
900	0.820	0.790	0.770	0.780	0.798
1000	0.926	0.902	0.868	0.874	0.876
1100	1.048	0.986	0.962	0.968	0.968
1200	1.078	1.068	1.046	1.044	1.038
1300	1.164	1.130	1.144	1.124	1.154
1400	1.344	1.222	1.192	1.224	1.244
1500	1.406	1.316	1.292	1.332	1.330
1600	1.484	1.402	1.384	1.376	1.398
1700	1.526	1.550	1.476	1.486	1.498
1800	1.606	1.610	1.562	1.618	1.592
1900	1.742	1.684	1.642	1.650	1.646
2000	1.798	1.726	1.760	1.714	1.758

Table 5.3: Experiment Results of Computational Time for *Exp.3*

First of all, we look at the three tables separately. In Table 5.1, the computational time grows pretty stable for *test1*, *test2* and *test3* from $N = 100$ to $N = 2000$, but for *test4* and *test5*, the computational time grows very fast. The reason is that the stairs' heights change very often and the agent itself can only recognize exactly the same situation for macro-actions, therefore, for the similar situations like

$$do([liftTill(h), forwLowLegS, stepDownS(main), moveBarycenterS(main),$$
$$straightLeg], s_0) \quad (5.7)$$

when h changes, the agent has to develop and store the new $maxPossBase$ for macro-action *stepSupp* in the situation instance. Hence, the more the different values of h have, the bigger the dynamic part of the knowledge base is, and the larger the time is to be consumed on finding all the possibilities of outcomes. In Table 5.2, we observed that the increasing of the computational time is very stable and almost the same for the five tests, that is to say, the change of the stair height doesn't affect the computational time. Analyzing the macro-action for procedure(3.7) *climbing*(h), we found that since for all the legal stairs' heights, they are in the same ob-class for macro-action *climbStair*(h), therefore, in the dynamic part of its knowledge base, the agent only need to keep one fact of $maxPossBase$ for macro-action *climbStair*(h) in the situation S_0 and this part won't change with the changing of height h. In Table 5.3, since there is no knowledge base at all for the *stGolog*, the changing of the stairs' heights certainly won't affect the computational time at all.

Secondly, we compare the three tables vertically. It is easy to see that the results in *Exp.2* is always better than in *Exp.3* for every test, and *Exp.1* has much better results than both *Exp.2* and *Exp.3* for *test1*, *test2* and *test3* when N becomes larger. The reason is that for the first three tests, we don't change the stairs' heights very often and moreover the relative frequency of the change of the stairs' heights to the number of the stairs N becomes even smaller when N becomes larger. But for *test4* and *test5*, *Exp.2* and *Exp.3* have much better results than *Exp.1*. The reason is very clear – the growth of the size of the dynamic part of the knowledge base consumes the time tremendously. Therefore, when the stair's height h changes very often, either $Exp.2$ or $Exp.3$ would be a better choice.

At last, comparing the three tables horizontally, especially for *test1*, *test2* and *test3*, it is easy to see *Exp.1* is a better choice. Analyzing the reason why *Exp.2* is not as competitive as *Exp.1* for the former three tests, we find that the maximal length of the macro-actions in *Exp.1* is much shorter than that of the macro-actions in *Exp.2*. According to the development of the static part of a knowledge base, *Exp.1* contains much less extended successor state axioms, therefore has smaller static part. Hence, when the dynamic part of the knowledge base is relatively stable, we would prefer to define short macro-actions.

5.2.3 Summary: the Benefits and the Limitations

After above discussion, we can clearly see the computational benefit of using macro-actions. Moreover, we somehow make the agent have some "memories", therefore can keep its "experience". The limitations of using macro-actions, especially under our current implementation, the duty of the controllers becomes heavier. It is the controller's responsibility to choose proper macro-actions and to make sure to change the dynamic part of the knowledge base when objects of different ob-classes for certain macro-actions occur. Finally, up till now, the agent can not be aware of similar local situations like (5.7) for different h's, to which the reuse of the macro-actions might also can be extended in our imagination. This limitations might open a door for part of our future work.

Chapter 6

Conclusions and Future Work

To make the autonomous agents perform more and more intelligently is always a goal for the AI researchers. In this paper, under the background of uncertainty systems, we introduced a concept of macro-action based on the existing complex Golog sequential action constructor, extended the basic action theories with macro-actions, defined an extended regression operator to help us to develop the knowledge base for the macro-actions in advance and later use the saved knowledges in application. Therefore, we somehow partially simulate the behaviors that the agents can learn knowledge in advance, keep the experience and later retrieve the existing experience when they meet the same situation. For implementation of developing the static part of a knowledge base, a program named *developer* was established. We also gave an interpreter named *macGolog* which was modified from the stGolog interpreter on the purpose of using and reusing the macro-actions with their knowledge base for a system. We discussed the advantages and limitations of introducing macro-actions into uncertainty systems, and concluded that the reuse of macro-actions can bring us computational benefit under the condition that the agent is in an environment that it needs to perform similar works in same local situations.

We certainly must mention the fountainhead of the idea of introducing macro-actions. This idea is adopted from using and reusing local policies [8] in solving the hybrid Markov

Decision Processes(hybrid MDPs)[8, 13, 21, 32] in the decision-making theory. Although we did not have chance to go that far to try to solve hybrid MDPs in high-level programming language yet, the researchers have begun their work on solving the MDPs and first-order MDPs in the situation calculus [4, 3].

There are many aspects remain open and are interesting to pursue in the near future.

1. There are several things can be improved during our implementation of using and reusing of macro-actions. First of all, we only assume that the agent can somehow switch to the local initial situations, and in the experiments we achieved this only by reseting the situations according to the controller's commands. To make the modeled systems more practical and more expressive, we may consider describing the exogenous interrupt actions by the controllers, or we may can set sensors for the autonomous agents so that they can switch the situations themselves once they sense some particular signals. Second, since we only focus on local situations, the agent will "forget" its former choices and performance of deterministic actions when the situation is reset. In later work, we may consider to somehow keep the former histories, therefore, we can also have global information and histories of reusing macro-actions if the agent or the controllers would prefer to.

2. As we discussed in previous chapter, the agent can only recognize the exact situations for reusing macro-action on the objects in same ob-class. And we observed that the situations like (5.7) for different h are very similar (actually can be viewed as local initial situations for macro-action *stepSupp*). In the future, we would like to do research on how to make the robot aware of these kinds of similar situations, so that we can reduce the size of the dynamic part of the knowledge base. Moreover, the solving of this question may even lead to the solutions of keeping the global histories of re-performing macGolog programs.

3. We also mentioned that our idea is adopted from the using of local policies in solving

hybrid MDPs. On the other hand, we are interested in studying the common and different characters of macro-actions in the this paper and the local policies in the MDPs, and consider the possibilities of solving hybrid MDPs by using the high-level programming languages like the situation calculus.

Bibliography

[1] *The Cognitive Robotics Group*. http://www.cs.utoronto.ca/cogrobo/.

[2] A. Bonner and M. Kifer. Transaction logic programming. Technical report, Dept. of Computer Science, University of Toronto, 1993.

[3] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for solving first-order markov decision processes. In G. Lakemeyer, editor, *Proceedings of Seventeenth International Joint Conference on Artificial Intelligence*, pages 690–970, Seattle,Washington, 2001.

[4] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of Seventeenth National Conference on Artificial Intelligence*, pages 355–362, 2000.

[5] D. Crevier. *AI : the Tumultuous History of the Search for Artificial Intelligence*. New York, NY, 1993.

[6] E. Davis. *Representations of Commonsense Knowledge*. Morgan Kaufmann Publishers, San Francisco, CA, 1990.

[7] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1221–1226, 1997.

[8] T. Dean and S.H. Lin. Decomposition techniques for planning in stochastic domains. *IJCAI-95*, pages 1121–1127, 1995.

[9] C. Elkan. Reasoning about action in first-order logic. In *Proc. of the Ninth Biannual Conf. of the Canadian Society for Computational Studies of Intelligence (CSCSI'92)*, pages 221–227, San Francisco, CA, 1992. Morgan Kaufmann Publishers.

[10] F.Bacchus. *Representing and Reasoning with Probabilistic Knowledge*. MIT Press, 1990.

[11] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[12] A.R. Haas. The frame problem in artificial intelligence. In F.M. Brown, editor, *Proceedings of the 1987 workshop*, pages 343–348, San Francisco, CA, Los Altos, California, 1987. Morgan Kaufmann Publishers.

[13] M. Hauskrecht, N. Meuleau, L. Kaelbling, T. Dean, and C. Boutilier. Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth Conference on Uncertainty In Artificial Intelligence*, pages 220–229, San Francisco, 1998. Morgan Kaufmann.

[14] R.A. Kowalski and M.J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[15] H. Levesque, R. Reiter, Y. Lespérance, Fangzhen Lin, and R.B. Scherl. Golog: A logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31(1-3):59–83, 1997.

[16] H. Levesque, R. Reiter, Y. Lespérance, Fangzhen Lin, and R.B. Scherl. Golog: A logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31(1-3):59–83, 1997.

[17] J. McCarthy, , and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.

[18] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[19] J. McCarthy. What is artificial intelligence. In *http://www-formal.stanford.edu/jmc/whatisai/whatisai.html*, September 2001.

[20] J. McCarthy, M. L. Minsky, N. Rochester, and C.E. Shannon. The proposal for the dartmouth summer research project on artificial intelligence. *http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html*, 1955.

[21] R. Parr. Flexible decomposition algorithms for weakly coupled markov decision problems. In *Proceedings of the Fourteenth Annual Conference in Uncertainty in Artificial Intelligence (UAI-98)*, pages 422–430. Morgan Kaufmann, 1998.

[22] E.P.D. Pednault. Adl:exploring the middle ground between strips and the situation calculus. In R.J. Brachman, H.J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning(KR'89)*, pages 324–332, San Francisco, CA, 1989. Morgan Kaufmann Publishers.

[23] E.P.D. Pednault. Adl and the state-transition model of action. *J. Logic and Computation*, 4(5):467–512, 1994.

[24] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–364, 1999.

[25] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Eiley & Sons, Inc., New York, NY, 1994.

[26] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76, New York, 1978. Plenum Press.

[27] R. 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, San Diego, 1991. Academic Press.

[28] R. Reiter. *Knowledge in Action*, chapter 12. The MIT Press, 2001.

[29] E. Sandewall. *Features and Fluents: A Systematic Approach to the representation of Knowledge about Dynamical Systems*. Oxford University Press, 1994.

[30] L.K. Schubert. Monotonic solution to the frame problem in the situation calculus: A efficient method for worlds with fully specified actions. In H.E. Kyberg, R.P. Louis, and G.N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67, Boston, Mass., 1990. Kluwer Academic Press.

[31] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[32] R.S. Sutton, Precup D., and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Journal of Artificial Intelligence*, 112:181–211, 1999.

[33] R. Waldinger. Achieving several goals simultaneously. In E.Elcock and D.Michie, editors, *Machine Intelligence 8*, pages 94–136, Edinburgh,Scotland, 1977. Ellis Horwood.

Appendix A

The following represents the stGolog interpreter given in [28] Chapter 12.

An stGolog Interpreter

```
:- set_flag(print_depth,100).

:- nodbgcomp.

:- dynamic(proc/2).            % Compiler directives. Be sure

:- set_flag(all_dynamic, on).  % that you load this file first!


:- op(800, xfy, [&]).   % Conjunction

:- op(850, xfy, [v]).   % Disjunction

:- op(870, xfy, [=>]).  % Implication

:- op(880,xfy, [<=>]).  % Equivalence

:- op(950, xfy, [:]).   % Action sequence


stDo(nil,1,S,S).

stDo(A : B,P,S1,S2) :- stochastic(A),

   (not (choice(A,C), poss(C,S1)), !,  % Program can't continue.

    S2 = S1, P = 1 ;                   % Create a leaf.

  % once is an Eclipse Prolog built-in. once(G) succeeds the first time

  % G succeeds, and never tries again under backtracking. We use it here

  % to prevent stDo from generating the same leaf situation more than
```

```
  % once, when poss has multiple solutions.
    choice(A,C), once(poss(C,S1)), prob(C,A,S1,P1),
    stDo(B,P2,do(C,S1),S2), P is P1 * P2 ).
stDo((A : B) : C,P,S1,S2) :- stDo(A : (B : C),P,S1,S2).
stDo(?(T) : A,P,S1,S2) :- holds(T,S1), !, stDo(A,P,S1,S2) ;
                          S2 = S1, P = 1. % Program can't continue.
                                          % Create a leaf.
stDo(if(T,A,B) : C,P,S1,S2) :- holds(T,S1), !, stDo(A : C,P,S1,S2) ;
                               stDo(B : C,P,S1,S2).
stDo(A : B,P,S1,S2) :- proc(A,C), stDo(C : B,P,S1,S2).
stDo(while(T,A) : B,P,S1,S2) :- holds(T,S1), !,
                                stDo(A : while(T,A) : B,P,S1,S2) ;
                                stDo(B,P,S1,S2).

prob(C,A,S,P) :- choice(A,C), poss(C,S), !, prob0(C,A,S,P) ; P = 0.0 .

stochastic(A) :- choice(A,N), !.

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)).
```

Appendix B

Since the concept of *uniform* ([28] Definition 4.4.1) appears in the literature frequently, to make it easy for the readers, we present the definition in this appendix, and also give sample proof that the regression of s-regressable formula is uniform in s.

Definition Uniform Formulas ([28] Definition 4.4.1)

Let σ be a term of sort situation. *Inductively define the concept of a term of $\mathcal{L}_{sc}$ that is* uniform in σ *as follows:*

1. *Any term that does not mention a term of sort* situation *is uniform in σ.*
2. *If g is an n-ary non-fluent function symbol, and $t_1, \cdots, t_n$ are terms that are uniform in σ and whose sorts are appropriate for g, then $g(t_1, \cdots, t_n)$ is uniform in σ.*
3. *If f is an $(n+1)$-ary functional fluent symbol, and $t_1, \cdots, t_n$ are terms that are uniform in σ and whose sorts are appropriate for f, then $f(t_1, \cdots, t_n, \sigma)$ is uniform in σ.*

The formulas of $\mathcal{L}_{sc}$ that are uniform in σ *are inductively defined by:*

1. *Any formula that does not mention a term of sort* situation *is uniform in σ.*
2. *When F is an $(n+1)$-ary relational fluent symbol, and $t_1, \cdots, t_n$ are terms that are uniform in σ and whose sorts are appropriate for F, then $F(t_1, \cdots, t_n, \sigma)$ is uniform in σ.*
3. *If U_1 and U_2 are formulas uniform in σ, so are $\neg U_1$, $U_1 \wedge U_2$ and $(\exists v)U_1$ provided that v is a variable not of sort* situation.

Property 1. *Suppose s is either S_0 or a variable of sort situation and the regression operator $\mathcal{R}$ is defined as Definition 3.6 in Chapter 3, for any s-regressable formula W in $\mathcal{L}_{sc}$, we have $\mathcal{R}[W]$ is a formula uniform in s.*

Proof: It can be proved by induction on the maximal length n of all the logs $[a_1, a_2, \cdots, a_m]$ satisfying $do([a_1, a_2, \cdots, a_m], s)$ in formula W (since n is the maximal length, we have $m \leq n$).

Base Case: $n = 0$, i.e. s is the only term of sort situation (if any) in W, then according to the definition of $\mathcal{R}$, $\mathcal{R}[W] = W$ which is definitely uniform in s.

Induction Steps: Let k be some arbitrary natural number, and suppose the proposition is true for all n such that $0 \leq n \leq k$, now we are going to prove it is true for $n = k + 1$.

According to the recursive definition of $\mathcal{R}$ when W is not atomic, it is sufficient to prove the proposition for atoms that include logs of length $n = k + 1$, since for all other atoms that include logs no longer than k, their regression results are uniform in s according to the hypothesis. Atoms that include logs of length $n = k + 1$ appear to have three cases:

(a) the atom is a relational fluent $F(\vec{x}, do([a_1, a_2, \cdots, a_{k+1}], s))$;

(b) the atom is a functional fluent $g(\vec{x}, do([a_1, a_2, \cdots, a_{k+1}], s))$; or

(c) the atom is of form $Poss(\vec{x}, do([a_1, \cdots, a_{k+1}], s))$.

For case (a), suppose F's successor state axiom in $\mathcal{D}_{ss}$ be

$$F(\vec{t}, do(a, s_1)) \equiv \Phi_F(\vec{t}, a, s_1) \ .$$

Without loss of generality, assume that all quantifiers (if any) of $\Phi_F(\vec{t}, a, s_1)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $F(\vec{x}, do([a_1, a_2, \cdots, a_{k+1}], s))$. Then

$$\begin{aligned} & \mathcal{R}[F(\vec{x}, do([a_1, a_2, \cdots, a_{k+1}], s))] \\ = \ & \mathcal{R}[\Phi_F(\vec{x}, a_{k+1}, do([a_1, a_2, \cdots, a_k], s))] \\ = \ & \Phi'_F(\vec{x}, s) \text{ for some } \Phi'_F, \end{aligned}$$

where $\Phi'_F(\vec{x}, s)$ is uniform in s according to the hypothesis. for the other two cases, the proof is similar according to the definition of $\mathcal{R}$.

Therefore, we proved for any s-regressable formula W in $\mathcal{L}_{sc}$, $\mathcal{R}[W]$ is uniform in s.
■

Similarly, we can prove following property for the extended operator $\mathcal{R}^*$ by using induction proof.

Property 2. *Suppose s is either S_0 or a variable of sort situation, the regression operator $\mathcal{R}^*$ is defined as Definition 4.2 in Chapter 4, for any s-regressable formula W in $\mathcal{L}'_{sc}$, we have $\mathcal{R}^*[W]$ is a formula uniform in s.*

Appendix C

The following represents the implementation of the developer of the knowledge base(static part) in Prolog.

The Developer in Prolog

```
% We assume that only variable S is used for representing current situation
% in this program, and S is not used for other attribution


:- set_flag(print_depth,100).
:- set_flag(all_dynamic, on).


:- op(900,xfy,[==>]).   % Simplification Rules.
:- op(800, xfy, [&]).   % Conjunction
:- op(850, xfy, [v]).   % Disjunction
:- op(870, xfy, [=>]).  % Implication
:- op(880,xfy, [<=>]).  % Equivalence
:- op(950, xfy, [:]).   % Action sequence


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% The Following part includes the main program of developer
```

```
kbDeveloper(L,File1,File2):-
       currentMaxLength(N1),  % get current maximal length of macro-actions
       newMaxLength(L, N2),   % get the maximal length of new macro-actions
       outputMac(L, File1, File2),
      (N1 < N2, 1 < N1, !,
       addSSA(N1,N2,File1,File2); % add new extended successor state axioms
       N1 < N2, N1 =< 1, !,
       addSSA(1,N2,File1,File2);
       N1 >= N2, !),
       addPoss(L,File1,File2),  % add new extended preconditions
       localPossProb(L,File2).  % add new localChoice and probMac_0

currentMaxLength(N):- setof(M, A^B^(macro(A,B),seqLength(B,M)), MS),
                      maxElement(MS,N).

maxElement([A],A):- number(A), !.
maxElement([A,B|T],N):- A >= B, !, N=A;
                        A < B, reverse([A,B|T],T1),maxElement(T1,N).

newMaxLength(L, N):- setof(M, A^B^(member([A,B],L),seqLength(B,M)), MS),
                     maxElement(MS,N).

% addSSA(N1,N2,File1,File2): compute the new extended successor state
% axioms for actions from length N1+1 to N2, and record the results
% into File1 and File2.

addSSA(N1,N2,File1,File2):- N0 is N1+1, addSSA0(N0, N2, R),
                            printRule(R,File1), printApp(R,File2).
addSSA0(N1,N2,R):- genNum(N1,N2,List),
```

```
        setof(F, B^C^restoreSitArg(F,B,C),FS), addSSA1(List,FS,R).
genNum(N,N,[N]):- !.
genNum(N1,N2,[N1|T]):- N1<N2, !, N3 is N1+1, genNum(N3,N2,T).
addSSA1([],_,[]):- !.
addSSA1([N|T], Fluents, R):- genNew('A',N,Actions),
        addSSA2(Fluents, Actions, R0), addSSA1(T, Fluents, R1),
        append(R1,R0,R).
addSSA2([],_,[]):- !.
addSSA2([F|T],Actions,[R1|T1]):- regression(F,Actions,R1),
                                 addSSA2(T,Actions,T1).

% addPoss(L,File1, File2): add extend precondition axioms for new sequences of
% deterministic actions; moreover, record the list of deterministic sequences,
% whose extended precondition axioms has been recorded into File2, into File1
% so that we can avoid duplication later while calling developer again.

addPoss(L,File1,File2):- addPoss0(L,Result1, Result2),
         removeFalse(Result1,Result3), printApp(Result3,File2),
        (not Result2 = [], !, printComputed(Result2,File1); Result2 = [], !).
removeFalse([],[]):- ! .
removeFalse([[_,false]|T],T1):- removeFalse(T,T1).
removeFalse([[A,B]|T],[[A,B]|T1]):- not B = false, !, removeFalse(T,T1).

addPoss0([],[],[]):- !.
addPoss0([[_,B]|T],R1,R2):- addPoss0(T,L1,M1), extPoss(B,L2,M2),
                            append(L2,L1,R1), append(M2,M1,R2).
extPoss(B,R1,R2):- setof(Act,
         N^(choiceMac(B,Act),seqLength(B,N),length(Act,N)), List),
         extPoss0(List,R1,R2).
```

```
extPoss0([],[],[]):- !.
extPoss0([H|T],R1,R2):- extPoss1([],H,L1,M1), extPoss0(T,L2,M2),
                        append(L1,L2,R1), append(M1,M2,R2).
extPoss1(_,[],[],[]):- !.
extPoss1(A,[H|T],R1,R2):- A=[], !, append(A,[H],A1),
                          extPoss1(A1,T,R1,R2).
extPoss1(A,[H|T],R1,R2):- not A=[], append(A,[H],A1),
         computedBefore(List),    % If poss(A1,S) has computed before
         member(A1,List), !,      % this time of calling developer,
         regression(A1,A,H,_),    % don't record the regression result
         extPoss1(A1,T,R1,R2).    % into File2.

extPoss1(A,[H|T],R1,R2):- not A=[], append(A,[H],A1),
         poss(A1,_) <=> _ , !,    % If poss(A1,S) has computed this time for
         extPoss1(A1,T,R1,R2).    % other macro-actions, don't compute again.

extPoss1(A,[H|T],[D|L1],[A1|M1]):- not A=[], append(A,[H],A1),
         not (computedBefore(List), member(A1,List);
         poss(A1,_) <=> _ ), !,                    % Otherwise,
         regression(A1,A,H,D), extPoss1(A1,T,L1,M1). % compute and record.

% localPossProb(L,File2): gather the deterministic choices of macro-action
% which is possible at situation S, compute regression results for macProb0,
% and output to database File2.

localPossProb(L,File2):- localPossProb0(L,Result1,Result2),
         printLocal(Result1,File2), printApp(Result2,File2).
localPossProb0([],[],[]):- !.
localPossProb0([[N,B]|T], [[N,AS1]|L1],R):- localPossProb0(T,L1,R1),
```

```
    setof(A, S^(choiceMac(B,A),once((length(A,1); not poss(A,S) <=> false))),
          AS),
    reverse(AS,AS1),           % make list arranged from longest to shortest
    macProbAction0(AS,N,R2), append(R2,R1,R).
macProbAction0([],_,[]):- !.
macProbAction0([A|T],N,[R|T1]):- regression(probMac0(A,N,S,Pr), R),
                                 macProbAction0(T,N,T1).



% recursive definition of choiceMac, to check whether a sequence of
% deterministic actions is a nature's choice of certain macro-action.

choiceMac(A1:A2,[C1|C2]):- choice(A1,C1), choiceMac(A2,C2).
choiceMac(A:_,[C]):- choice(A,C).
choiceMac(A,[C]):- choice(A,C).

% compute the length of macro-action,
seqLength(A,1) :- not A =.. [:|_], !.
seqLength(_:B,N) :- seqLength(B,N1), N is N1+1.

% special database to avoid no definitions of "macro" and "computedBefore"
% at initial situation.
macro(nil, nil).
computedBefore([]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% regression procedures

regression(F,A,[F1,R]):- restoreSitArg(F,do(A,S),F1),
```

```
         lastElement(A,M1), formerActions(M0,[M1],A),
         restoreSitArg(F,do([M1],do(M0,S)),F2),
         regress1(F2,R), asserta((F1 <=> R)). % for extended successor state
regression(F,[F,R]):-  regress2(F,R).         % for probabilities probMac_0
regression(A1,A,H,[poss(A1,S),D1]):-          % for extended preconditions
         regress1(poss(H,do(A,S)), C1),
         (length(A,1), !, member(M,A), poss(M,S) <=> R1;
          not length(A,1), !, poss(A,S) <=> R1),
         (R1 = true, !, D1 = true;
          R1 = false, !, D1 = false;
          not (R1 = true; R1 = false), !,simplify(R1 & C1, D1)),
         asserta((poss(A1,S) <=> D1)).

lastElement([E],E):- !.
lastElement([_|B],E):-lastElement(B,E).

formerActions(A,B,C):- append(A,B,C).

regress2(probMac0([C],A,S,Pr),R):- length([C],1), !, getAction(A,1,A1),
         regress1(prob0(C,A1,S,Pr),R), asserta(probMac0([C],A,S,Pr) <=> R).
regress2(probMac0(C,A,S,Pr),R):- length(C,N), N>1, !,
         lastElement(C,C1), formerActions(C2,[C1],C),
         getAction(A,N,AN), probMac0(C2,A,S,Pr1) <=> R1,
         regress1(prob0(C1,AN,do(C2,S),Pr2), R2),
         simplify(R1 & R2 & (Pr is Pr1 * Pr2),R),
         asserta(probMac0(C,A,S,Pr) <=> R).


% get the Nth primitive action E in a sequence of actions A
```

```
getAction(A,N,E):- macro(A,B), !, getStochastic(B,N,E).
getAction(A:B,N,E):- getStochastic(A:B,N,E).

getStochastic(E:_,1,E):- !.
getStochastic(E,1,E):- not E =.. [:|_], !.
getStochastic(_:E2,N,E):- N>1, !, N1 is N-1,
                            getStochastic(E2,N1,E).

% regress1(A,R): for term A, if A atom, and of form f(...,do(...,S)),
% we will find out the rule whose head matches A,
% and we will rename the quantified variable in the rule,
% then get the equivalent formula for A and
% do regression on the equivalent formula to get R.

regress1(A,R):-
         matching(A,Head), Head <=> Body, quant(Head, Body,LV12),
         term_variables(A, LA), sameVar(LV12, LA, LV2),
         (LV2=[], !, A <=> Body1;
          not LV2=[], !, term_variables([A,Body],LV1),
          genNew(LV2,LV1,New),           % generate new variables
          rename(LV2,New,Body,BodyR),
          retract(Head <=> Body), asserta(Head <=> BodyR), A <=> Body1),
         simplify(Body1,Body2), regress(Body2,R).

sameVar(_, [], []):- !.
sameVar(A,[M|T],[M|T1]):- not newVar(M,A), !, sameVar(A,T,T1).
sameVar(A,[M|T],T1):- newVar(M,A), !, sameVar(A,T,T1).
```

```
regress(P & Q, R) :- regress(P,R1),
         (R1 = false, R = false, !; regress(Q,R2), simplify(R1 & R2,R)).
regress(P v Q, R) :- regress(P,R1),
         (R1 = true, R = true, !; regress(Q,R2), simplify(R1 v R2,R)).
regress(-P,R) :- regress(P,R1), simplify(-R1,R).
regress(A,R):- isAtom(A), not onlyS(A), A <=> _, !, regress1(A,R).
regress(A,R):- isAtom(A), (not A <=> _ ; onlyS(A)), !, R = 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)).

% onlyS(A): last augment of term A is current situation S.
onlyS(A):- functor(A,_,N), N>0, !, arg(N,A,Sit),
           get_var_info(Sit,name, R), R = 'S'.

% find the head of the rule which matches A
matching(A,Head):- functor(A,F,M),
           findall(B, W^( B <=> W, functor(B,F,M), matching0(A,B)), BS),
           member(Head, BS).
matching0(A,D):- A =.. [F|[B1|_]],
                (not F = prob0, not F = poss, !,
                 numActions(A,N),numActions(D,N);
                 (F = prob0;F = poss), !,
                 D =..[F|[B2|_]], functor(B1,F1,M1), functor(B2,F1,M1)).

numActions(A,N):- A =.. [_|B], reverse(B,[M|_]), M =.. [do,T,_], length(T,N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% notice that in rule Head <=> Body, variables which appear in Body,
```

```
% but not in Head, are quantified,
% quant(Head,Body,LV): LV is the list of quantified variables.
quant(Head,Body,LV):- term_variables(Head,V1),
               term_variables(Body,V2),
               diff(V2,V1,LV).

% diff(A,B,C): C = A-B
diff(L,[],L):- !.
diff(A,[A1|T],L):- newVar(A1,A), !, diff(A,T,L).
diff(A,[A1|T],L):- not newVar(A1,A), !,
                   removeElement(A1,A,B), diff(B,T,L).

removeElement(_,[],[]):- !.
removeElement(A1,[A|B],T):- get_var_info(A1, name, R1),
                 get_var_info(A, name, R1), removeElement(A1,B,T).
removeElement(A1,[A|B],[A|T]):- get_var_info(A1, name, R1),
                 not get_var_info(A, name, R1), removeElement(A1,B,T).

newVar(A,[]):- var(A), !.
newVar(A,[B|T]):- get_var_info(B, name, R2), get_var_info(A, name, R1),
                  not R1=R2, !, newVar(A,T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% rename(Names,News,Term1,Term2): Term2 is Term1 with Names
% replaced by corresponding News.

rename([],[],A, A):- !.
rename([X|T1],[Y|T2],A,B):- sub0(X,Y,A,B1), rename(T1,T2,B1,B).
```

```
% sub0(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New.
sub0(X1,_,T1,T1) :- var(T1), not var(X1).
sub0(X1,_,T1,T1) :- var(T1), var(X1), get_var_info(X1, name, R1),
                    not get_var_info(T1, name, R1).
sub0(X1,X2,T1,X2) :- var(T1), var(X1), get_var_info(X1, name, R1),
                     get_var_info(T1, name, R1).
sub0(X1,X2,T1,T2) :- not var(T1), not var(X1), T1 = X1, T2 = X2.
sub0(X1,X2,T1,T2) :- not var(T1), not var(X1),
                     not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                     T2 =..[F|L2].
sub0(X1,X2,T1,T2) :- not var(T1), var(X1), T1 =..[F|L1],
                     sub_list(X1,X2,L1,L2),T2 =..[F|L2].

sub_list(_,_,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub0(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simplify formulas.
simplify(-(-W),S) :- simplify(W,S), !.
simplify(W1 & W2,S) :- simplify(W1,S1), simplify(W2,S2),
                       simplify1(S1 & S2,S), !.
simplify(W1 v W2,S) :- simplify(W1,S1), simplify(W2,S2),
                       simplify1(S1 v S2,S), !.
simplify(-W,S) :- simplify(W,S1), simplify1(-S1,S), !.
simplify(A,S) :- simplify1(A,S).

simplify1(W,Simp) :- W ==> Simp, !.
simplify1(W,W).
```

```
%  Simplification Rules.


true & P ==> P.          P & true ==> P.       false & _ ==> false.

_ & false ==> false.     true v _ ==> true.    _ v true ==> true.

false v P ==> P.         P v false ==> P.      -true ==> false.

-false ==> true.

X v - Y ==> true :- not var(X), not var(Y), X == Y.

- X v Y ==> true :- not var(X), not var(Y),  X == Y.

X & - Y ==> false :- not var(X), not var(Y), X == Y.

- X & Y ==> false :- not var(X), not var(Y), X == Y.

X & Y ==> X :- not var(X), not var(Y), X == Y.

X v Y ==> X :- not var(X), not var(Y), X == Y.

X = Y ==> true :- not var(X), not var(Y), matchWith(X,Y).

X = Y ==> false :- not var(X), not var(Y), not X=Y.


matchWith(X,Y):- var(X), not var(Y),!, fail.

matchWith(X,Y):- not var(X), var(Y), Y = X, !.

matchWith(X,Y):- var(X), var(Y), Y = X, !.

matchWith(X,Y):- not var(X), not var(Y), !,

                 X =.. [F|L1], Y=.. [F|L2], match_list(L1,L2).

match_list([],[]).

match_list([A1|B1],[A2|B2]):- matchWith(A1,A2),match_list(B1,B2).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% we always use file "variables" to be the temporary files to help us to

% generate new variables we need

genNew(V,N,L):- genVarFile(V,N), getVarFile(L), delete(variables).


genVarFile(V,N):-
```

```
          (number(N), !, gensym(V,N,T);
           not number(N), !, gensym0(V,N,T)),
          flaten(T,T1),open(variables,write, Stream),
          printf(Stream, "newVar(%w).", [T1]), close(Stream).

% gensym(V,N,T): generate list of strings with prefix V
% followed by n, n=1,2,...,N
gensym(_,0,[]):- !.
gensym(V,N,R):- N>0, N1 is N-1, gensym(V,N1,T),
                concat_atom([V,N],A), append(T,[A],R).

% gensym0(A,B,T): every element in T is a string with exact one prefix
% in A, and all elements in T are different from variables in B.
gensym0([],_,[]):- !.
gensym0([A|T1], B, [A1|T2]):- get_var_info(A, name, R),
             initializeVarCount, genEle(R,B,A1), gensym0(T1, B, T2).

genEle(A,B,A1):- getval(varCount,N), concat_atom([A,N],T),
             (not haveSameName(T,B), !, A1=T;
              haveSameName(T,B), !, incrementVarCount, genEle(A,B,A1)).

initializeVarCount:- setval(varCount,0).
incrementVarCount:- incval(varCount).

haveSameName(A,[B|_]):- get_var_info(B,name,A), !.
haveSameName(A,[B|T]):- not get_var_info(B,name,A), haveSameName(A,T).

% make ['A','B'] look like 'A,B'
flaten([],A):- A='',!.
```

```
flaten([A|B],T):-
     length([A|B], N), N=1,!, flaten(B,T1),concat_atom([A,T1],T).
flaten([A|B],T):-
     length([A|B], N), N>1,!, flaten(B,T1),concat_atom([A,',',T1],T).



getVarFile(L):- getFromFile(L1), getVar(L1,L).


getFromFile(L):- open(variables, read, Stream),
                 readvar(Stream,_,L), close(Stream).


% getVar(L1,L): L1 is a list of elements of form [A|B],
% where A is name of variable B, L is lists of B's
getVar([],[]):- !.
getVar([[_|B]|T1],[B|T]):- var(B), getVar(T1,T).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% various of output and printing procedures


% add new macro-actions
outputMac(L, File1, File2):- printMacro(L,File1),
      printMacro(L,File2), assertMacro(L).


assertMacro([]).
assertMacro([[Name,Body]|T]):- not macro(Name,Body), !,
                               assert(macro(Name,Body)), assertMacro(T).


printMacro(L,File):- open(File, append, Stream),
                     printMacro0(Stream,L), close(Stream).
```

```
printMacro0(Stream,[]):- nl(Stream), !.
printMacro0(Stream,[[Name,Body]|T]):-
      nl(Stream), printf(Stream, "macro(%w,%w).", [Name,Body]),
      nl(Stream), printMacro0(Stream, T).

% printComputed(L,File): print fact computedBefore(L) into File
printComputed(L,File):- open(File, append, Stream),
      nl(Stream), printf(Stream, "computedBefore(%w).", [L]),
      nl(Stream), close(Stream).



% printLocal(L,File): print ‘‘localChoice’’ to File
printLocal(L,File):- open(File, append, Stream),
      printLocal0(Stream,L), close(Stream).

printLocal0(Stream,[]):- nl(Stream), !.
printLocal0(Stream,[[Name,Body]|T]):- nl(Stream),
      printf(Stream, "localChoice(%w,%w).", [Name,Body]),
      nl(Stream), printLocal0(Stream, T).

% printRule(T,Tempbase): output new rules to file Tempbase which
% is a file save the extended successor state axiom in form of
% H <=> W, and this Tempbase helps the program "developer" to g
% enerate new results for new macro-action.
printRule(T,Tempbase):- open(Tempbase, append, Stream),
                        printall0(Stream, T), close(Stream).

printall0(_,[]):- !.
printall0(Stream,[[Head, Body]|T]):-
```

```
      nl(Stream), writeclause(Stream, (Head <=> Body)),
      nl(Stream), printall0(Stream,T).



% printApp(L,Appbase): output new rules to file Appbase which
% is a file save the extended axioms in clause form, and this
% Appbase is used in application for data reuse.

printApp(L,Appbase):- open(Appbase,append,Stream),
                      printall1(Stream,L), close(Stream).

printall1(Stream,[]):- nl(Stream), !.

printall1(Stream,[[Head, Body]|Tail]):-
      nl(Stream), term_string(Head,Head1),
      printf(Stream, "%w :- ", [Head1]),
      printSentence(Stream,Body), printf(Stream, ".",[]),
      nl(Stream), printall1(Stream,Tail).

printSentence(Stream, W):- isAtom(W), !, term_string(W,W1),
                           printf(Stream, " %w ",[W1]).
printSentence(Stream, -W):- isAtom(W), !, term_string(W,W1),
      printf(Stream, "%s", ["not "]), printf(Stream, " %w ",[W1]).
printSentence(Stream, -W):- not isAtom(W), !,
      printf(Stream, "%s", ["not ("]), printSentence(Stream, W),
      printf(Stream, "%s", [") "]).
printSentence(Stream, W1 & W2):- not isLiteral(W1), not isLiteral(W2), !,
      printf(Stream, "%s", [" ("]), printSentence(Stream,W1),
      printf(Stream, "%s", ["), ("]), printSentence(Stream,W2),
```

```
      printf(Stream, "%s", [") "]).
printSentence(Stream, W1 & W2):- isLiteral(W1), not isLiteral(W2), !,
      printSentence(Stream,W1), printf(Stream, "%s", [", ("]),
      printSentence(Stream,W2), printf(Stream, "%s", [") "]).
printSentence(Stream, W1 & W2):- not isLiteral(W1), isLiteral(W2), !,
      printf(Stream, "%s", [" ("]), printSentence(Stream,W1),
      printf(Stream, "%s", ["), "]), printSentence(Stream,W2).
printSentence(Stream, W1 & W2):- isLiteral(W1), isLiteral(W2), !,
      printSentence(Stream,W1), printf(Stream, "%s", [", "]),
      printSentence(Stream,W2).
printSentence(Stream, W1 v W2):- not isLiteral(W1), not isLiteral(W2), !,
      printf(Stream, "%s", [" ("]), printSentence(Stream,W1),
      printf(Stream, "%s", ["); ("]), printSentence(Stream,W2),
      printf(Stream, "%s", [") "]).
printSentence(Stream, W1 v W2):- isLiteral(W1), not isLiteral(W2), !,
      printSentence(Stream,W1), printf(Stream, "%s", ["; ("]),
      printSentence(Stream,W2), printf(Stream, "%s", [") "]).
printSentence(Stream, W1 v W2):- not isLiteral(W1), isLiteral(W2), !,
      printf(Stream, "%s", [" ("]), printSentence(Stream,W1),
      printf(Stream, "%s", ["); "]), printSentence(Stream,W2).
printSentence(Stream, W1 v W2):- isLiteral(W1), isLiteral(W2), !,
      printf(Stream, "%s", [" "]), printSentence(Stream,W1),
      printf(Stream, "%s", ["; "]), printSentence(Stream,W2).


isLiteral(W):- isAtom(W), !.
isLiteral(-W):- isAtom(W), !.
```

Appendix D

The following represents the complete interpretation of the example of robot climbing stairs in Prolog for the macGolog interpreter. The only difference from the original one in the stGolog is that we modify $do(A, S)$ to be $do([A], S)$.

Robot Climbing Stairs in Prolog for the macGolog

```
% Declare nature's choices
choice(liftUpperLeg(H),C):- C = liftTill(H); C = malfunc(H).
choice(forwLowLeg, C):- C =  forwLowLegS; C = forwLowLegF.
choice(stepDown(L), C):- C = stepDownS(L); C = stepDownF(L).
choice(moveBarycenter(L), C):- C = moveBarycenterS(L); C = moveBarycenterF(L).
choice(straightLeg, C):- C = straightLeg.
choice(forwSupLeg, C):- C =  forwSupLegS; C = forwSupLegF.

% Action precondition and successor state axioms
poss(liftTill(H),S) :- barycenter(supporting,S).
poss(malfunc(H),S) :- barycenter(supporting,S).
poss(forwLowLegS,S) :- not mainToCurr(wrongPos,S), not footOnGround(main,S).
poss(forwLowLegF,S) :- not mainToCurr(wrongPos,S), not footOnGround(main,S).
poss(stepDownS(L),S) :- not footOnGround(L,S), overNewStair(L,S).
poss(stepDownF(L),S) :- not footOnGround(L,S), overNewStair(L,S).
```

```
poss(moveBarycenterS(L),S) :- footOnGround(L,S).

poss(moveBarycenterF(L),S) :- footOnGround(L,S).

poss(straightLeg,S) :- not straightMain(S), footOnGround(main,S),
                       barycenter(main,S).

poss(forwSupLegS,S):- barycenter(main,S), straightMain(S).

poss(forwSupLegF,S):- barycenter(main,S), straightMain(S).

straightMain(do([A],S)):- A = straightLeg;
                          straightMain(S), not A = liftTill(H).

barycenter(L,do([A],S)):- A = moveBarycenterS(L);
     barycenter(L,S), not (A = moveBarycenterS(L1), not L=L1).

footOnGround(L,do([A],S)):- A = stepDownS(L);
     footOnGround(L,S), (L = main, not A = liftTill(H);
     L = supporting, not A = straightLeg).

overNewStair(L,do([A],S)):- A = forwSupLegS, L = supporting;
     A = forwLowLegS, L = main; overNewStair(L,S), not A = stepDownS(L).

mainToCurr(H,do([A],S)):- A = liftTill(H);
     A = malfunc(H1), H = wrongPos; A = stepDownS(main), H = 0;
     mainToCurr(wrongPos,S), H = wrongPos; mainToCurr(H,S),
     not H = wrongPos, not A= malfunc(H1), not A = stepDownS(main),
     not (A = liftTill(H1), not H = H1).


% Probabilities

prob0(liftTill(H),liftUpperLeg(H),S,Pr):- Pr is 100/(H+100).

prob0(malfunc(H),liftUpperLeg(H),S,Pr):- Pr is H/(H+100).

prob0(forwLowLegS,forwLowLeg,S,Pr):- mainToCurr(H,S), Pr is 80/(H+80).

prob0(forwLowLegF,forwLowLeg,S,Pr):- mainToCurr(H,S), Pr is H/(H+80).

prob0(stepDownS(L),stepDown(L),S,Pr):- Pr = 0.9.

prob0(stepDownF(L),stepDown(L),S,Pr):- Pr = 0.1.

prob0(moveBarycenterS(L),moveBarycenter(L),S,Pr):- Pr = 0.8.
```

```
prob0(moveBarycenterF(L),moveBarycenter(L),S,Pr):- Pr = 0.2.
prob0(straightLeg,straightLeg,S,Pr):- Pr = 1.0.
prob0(forwSupLegS,forwSupLeg,S,Pr):- Pr = 0.8.
prob0(forwSupLegF,forwSupLeg,S,Pr):- Pr = 0.2.

restoreSitArg(straightMain,S,straightMain(S)).
restoreSitArg(barycenter(L),S,barycenter(L,S)).
restoreSitArg(footOnGround(L),S,footOnGround(L,S)).
restoreSitArg(overNewStair(L),S,overNewStair(L,S)).
restoreSitArg(mainToCurr(H),S,mainToCurr(H,S)).
primitive_action(liftTill(_)).
primitive_action(malfunc(_)).
primitive_action(forwLowLegS).
primitive_action(forwLowLegF).
primitive_action(stepDownS(_)).
primitive_action(stepDownF(_)).
primitive_action(moveBarycenterS(_)).
primitive_action(moveBarycenterF(_)).
primitive_action(straightLeg).
primitive_action(forwSupLegS).
primitive_action(forwSupLegF).

% Initial Database
straightMain(s0).  mainToCurr(0,s0).
barycenter(supporting,s0).
footOnGround(L,s0):- L = main; L=supporting.
overNewStair(L,s0):- fail.
legalStair(H):- number(H), 0<H, H<20.
```