Macro-actions in the Stochastic Situation Calculus

Yilan Gu
Department of Computer Science
University of Toronto
Toronto, ON, M5S 3G4, Canada
yilan@cs.toronto.edu

Abstract

In this paper, we focus on the problem of modeling autonomous agents that perform similar strategies repeatedly in the same local situations in a stochastic system. We introduce the concept of macro-action into the situation calculus, extend the basic action theories and the regression operator, and develop a knowledge base for macro-actions so that the agent can remember certain details and later "recall" them when the agent performs the macro-actions in the same local situations again, thereby saving computational time.

1 Introduction

The *situation calculus* (SitCal) [McCarthy, 1963; Reiter, 2001] is a predicate logical language used to describe autonomous agents acting in a dynamic system. After solving the frame problem in the SitCal [Reiter, 1991], research on the applications of this language and further extensions of the language are now a very active area in artificial intelligence.

As we know, in the real world, people often meet with unpredictable situations. So it is rational to require autonomous agents to deal with stochastic outcomes of actions. Hence, from the late 1990s, based on the study of logic with probability [Bacchus, 1990] and the SitCal, researchers began to work on high-level programming for robots acting in stochastic systems. From different points of view, they offer several formalisms, including *stGolog* [Reiter, 2001], *dtGolog* [Boutilier *et al.*, 2000], etc.

The reasoning mechanism involved in former work is only based on primitive actions. For instance, by using the stGolog interpreter, we can determine how probable some state is after an agent performs a plan consisting of stochastic actions. The computation of probabilities and reasoning about effects of actions after executing the plan is reduced to reasoning about primitive actions by applying regression operator the is defined for primitive actions only [Reiter, 2001]. Even if this plan involves much repetition of sub-strategies, regression still needs to be performed step by step, which is inefficient and can be avoided. Often an autonomous agent works in a similar environment and is asked to solve similar problems, which involves lots of repetition in actions and outcomes. For example, if we ask a robotic agent to climb hundreds of continuous same-height stairs, it can be viewed as that the agent repeats a certain sequence of actions at the same local situation hundreds of times provided that we can reset the agent's situation to be some local initial situation whenever malfunctions occur. In this paper, we consider certain combinations of stochastic actions as a whole, called *macro-action*s, preprocess its properties, and later reuse the outcomes to save computational time. In fact, recently researchers proposed some related research. Greenwald [Greenwald, 1995] argued that it is reasonable to avoid reasoning in detail all the time when dealing with stochastic systems that must take time-critical actions. Castillo and Wrobel [Peña Castillo and Wrobel, 2002] proposed macro-operators in multirelational learning to reduce search space.

2 The Situation Calculus, Golog and StGolog

The basic ingredients of the language of the situation calculus $\mathcal{L}_{sc}$ includes actions, situations, objects and fluents. *Actions* are first-order terms representing actions in a dynamic world. *Situations* are first-order terms which denote possible world histories. A distinguished constant S_0 are used to denote the *initial situation*, and function $do(a, s)$ denotes the situation that results from performing action a in situation s. In fact, every situation corresponds to a sequence of actions. Moreover, binary relation $s \sqsubset s'$ represents that s is a proper sub-history of s'. *Objects* are a catch-all sort representing for everything else depending on the domain of application. *Fluents* are predicates and functions whose values may vary from situation to situation. By convention, the last argument of a fluent is a situation. $do([a_1, \cdots, a_n], s)$ is an abbreviation of $do(a_n, do(\cdots, do(a_1, s) \cdots))$ and $[a_1, \cdots, a_n]$ is called a *log*.

A *basic action theory* $\mathcal{D}$ is a set of axioms represented in $\mathcal{L}_{sc}$ with following five classes of axioms to model actions and their effects in a given dynamic system.

Action precondition axioms $\mathcal{D}_{ap}$: For each action function $A(\vec{x})$, there is one axiom of the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$. $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}$ and s, which characterizes the preconditions of action A.

Successor state axioms $\mathcal{D}_{ss}$: For each relational fluent $F(\vec{x}, s)$, there is a sentence in $\mathcal{L}_{sc}$ of form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is a formula with free variables among $\vec{x}, a$ and s, Similarly, a successor state axiom for a functional fluent $f(\vec{x}, s)$ is a formula of form $f(\vec{x}, do(a, s)) = y \equiv \Phi_f(\vec{x}, y, a, s)$. The successor state axiom for fluent F (respectively, f) completely characterizes the value of fluent F (respectively, f) in the next situation $do(a, s)$ in terms of the current situation s.

Initial database, denoted as $\mathcal{D}_{S_0}$: It is a set of first-order sentences whose only situation term is S_0; it specifies the initial problem state.

The rest two classes **Fundamental axioms** $\mathcal{D}_f$ and **Unique name axioms** $\mathcal{D}_{una}$ are mechanical and assumed to be true.

Regression is a central computational mechanism that forms the basis for automated reasoning in the situation calculus [Pednault, 1994; Reiter, 2001]. Reiter et al. give a recursive definition of a regression operator $\mathcal{R}$ on *regressable*[1] formula. 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 a stochastic system, *stochastic actions* are introduced. The outcomes of a stochastic action are nature's choices, i.e., not under the control of the robot. We characterize this by:

$$choice(\alpha, a) \overset{def}{=} a = A_1 \vee a = A_2 \vee \cdots \vee a = A_n,$$

where α is a stochastic action and A_i's are primitive actions. The axioms in $\mathcal{D}$ are presented for each A_i. The probability of each outcome of a stochastic action is presented as well. We require that whenever one of nature's action's preconditions is false, the action will have zero probability, i.e.,

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

Here, $prob_0(a, \beta, s)$ is provided by the controllers. It is axiomatizer's responsibility to ensure that a proper probability distribution has been defined while formalizing a probabilistic domain in $\mathcal{L}_{sc}$. One needs to verify two properties:
(a) $Poss(A_i, s) \supset prob_0(A_i, \alpha, s) > 0, \;\; i = 1, 2, \ldots, k$;
(b) $Poss(A_1, s) \vee \cdots \vee Poss(A_k, s) \supset \sum_{i=1}^{k} prob(A_i, \alpha, s) = 1$
for any stochastic action α and its nature's choices A_i.

Based on above extensions, new programs, named *stGolog* programs [Reiter, 2001] are constructed from stochastic actions together with the Golog [Levesque *et al.*, 1997] program constructors: *sequence* α;β – do action α followed by action β; *test action* $p?$ – test the truth value of expression p in the current situation; *conditionals if-then-else* and *while* loops; and *procedures, including recursion*. stGolog programs do *not* involve any form of nondeterminism: neither the *nondeterministic choice* of two actions, nor the π operator are allowed. Moreover, a dummy symbol *nil* is introduced into a sequence indicating the end of the sequence, which is another difference from Golog sequence. The stGolog interpreter is developed via $stDo(\alpha, p, s, s')$ meaning that agent performs stGolog program (or actions) α at the situation s, and 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 $\gamma$is defined: if $\mathcal{D}$ stands for the basic action theory,

$$probF(\psi, \gamma) \overset{def}{=} \sum_{\{(p,\sigma)\mathcal{D} \models stDo(\gamma:nil, p, S_0, \sigma) \wedge \psi(\sigma)\}} p.$$

3 Introducing Macro-actions

Often agents meet similar local situations, work on similar tasks and repeat the same strategies.

Example 1 Consider a robot with a $main$ leg and a supporting leg $spleg$ is asked to climb stairs (Figure 1). Climbing a stair can be viewed as performing a sequence of stochastic actions successfully after checking the stair height h is capable of climbing: the robot lifts its $main$ leg's upper part to height h, moves the $main$ leg's lower part forward, puts the foot down, moves its barycenter to $main$ leg, straightens it , then moves its leg $spleg$ forward, puts the foot down, and finally moves it barycenter back to leg $spleg$.

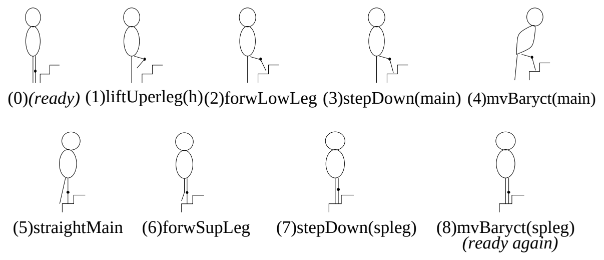


Figure 1: Action Decomposition of Robot Climbing Stairs

Provided that the controller can step in and reset the agent if a malfunction occurs, we can view the procedure of climbing n stairs as the procedure of repeatedly climbing one stair in same local situation n times. By using stGolog to reason about the probability of every possible outcomes of the procedure of a robot climbing hundreds of stairs, we need to perform regression step by step for every primitive action. Now, we expect that the agent can have some "memory" to save the intermediate results of climbing one stair, and later will "recall" the information, and therefore will perform climbing actions without "thinking" step by step again. To achieve this, our intuitive idea is to combine certain complex actions together, and the agents will compute and save its information in advance and later reuse it.

We focus on sequential action constructors and construct macro-action by sequence of stochastic actions in this paper. The rationale is that finite sequence of stochastic actions' characters is easy to be traced, and we can easily and formally develop extended axioms for sequential actions as shown next.

3.1 The Extended Action Theory

Definition 1 (s-regressable formula)
*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*

[1]A formula W of $\mathcal{L}_{sc}$ is **regressable** iff (1) every term of sort situation in W has the syntactic form $do([\alpha_1, \cdots, \alpha_n], S_0)$; (2) for every atom of the form $Poss(\alpha, \sigma)$ in W, α has the syntactic form $A(t_1, \cdots, t_n)$ for some n-ary function symbol A of $\mathcal{L}_{sc}$; and (3) W does not quantify over situations, and does not mention the relation symbols "$\sqsubset$" or "=" between terms of situation sort.

$do([a_1, \cdots, a_n], s)$ for some $n \geq 0$; (2) other conditions are same as the 2^{nd} and 3^{rd} conditions in footnote 1.

Notice that a regressable formula defined in [Reiter, 2001] is same as a S_0-*regressable* formula.[2] Naturally, we then can extend the regression operator $\mathcal{R}$ to an s-regressable formula W for some situation s.

Now we return to discuss how we can extend the axioms in a basic action theory and the probability axioms for macro-actions. Notice that the body of a macro-action is a sequence of actions, and its deterministic choices should also be sequential. Since performing a deterministic sequence in a particular situation ends up at a unique situation, we extend the notation $do(a, s)$ where a is of sort action to $do(a_1; \cdots ; a_n, s)$ $(n \geq 1)$ for primitive actions $a_1, \cdots, a_n$, indicating the situation after executing deterministic sequential action $a_1; \cdots ; a_n$ in the situation s. The purpose is to distinguish it from *log* and for later convenience.

Suppose there is a sequential action $A = A_1; \cdots ; A_n$, where each A_i is primitive deterministic action. The precondition axiom $Poss(A, s)$, meaning that it is possible to perform sequential action A in the situation s, can be extended as follows: $Poss(A, s)$ is the given axiom in $\mathcal{D}_{ap}$ if A is primitive; otherwise

$$Poss(A, s) \overset{def}{=} \quad \mathcal{R}[Poss(A_1, s) \wedge (\wedge_{i=2}^{n} Poss(A_i, do([A_1, \cdots, A_{i-1}], s)))],$$

which is a formula uniform in s, i.e., s is the only term of sort situation (if any) mentioned by the formula.

Given $a = a_1; \cdots ; a_n$, where each a_i is a variable of sort action, 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:
If $n = 1$, axiom $F(\vec{x}, do(a, s)) \equiv \phi_F(\vec{x}, a, s)$ (respectively, $f(\vec{x}, do(a, s)) = y \equiv \phi_f(\vec{x}, y, a, s)$) is the given successor state axiom in $\mathcal{D}$; otherwise,

$$F(\vec{x}, do(a, s)) \overset{def}{=} \mathcal{R}[F(\vec{x}, do([a_1, \cdots, a_n], s))]$$
$$= \psi_F(\vec{x}, a_1, \cdots, a_n, s) \text{ for some } \psi_F \text{ uniform in } s,$$
$$f(\vec{x}, do(a, s)) = y \overset{def}{=} \mathcal{R}[f(\vec{x}, do([a_1, \cdots, a_n], s)) = y]$$
$$= \psi_f(\vec{x}, y, a_1, \cdots, a_n, s) \text{ for some } \psi_f \text{ uniform in } s.$$

We can extend the probability function $prob$ for a deterministic sequential action $A = A_1; \cdots ; A_m$ and stochastic sequential action $\alpha = \alpha_1; \cdots ; \alpha_n$, denoted as $probMac$:

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

in which we define predicate $choiceMac$ as follows:

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

and say that A is a nature's choice for α if $choiceMac(\alpha, A)$ is true. In fact, $choiceMac$ is an extension of $choice$ and $probMac$ is an extension of $prob$.

To specify an appropriate probabilistic domain for a stochastic system, we need to verify that a proper probability distribution has been defined. Here, we can easily prove several properties for the definition of $probMac$ as follows.

Theorem 1 *Let $\alpha = \alpha_1; \alpha_2; \cdots ; \alpha_n$ be a stochastic sequential action, and A be a deterministic sequential action satisfying that $choiceMac(\alpha, A) \equiv true$. Suppose properties (a) and (b) in Section 2 have been verified, then*

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

Definition 2 *For any situation s and stochastic sequential action $\alpha = \alpha_1; \cdots ; \alpha_n$, we define the set $maxPoss(\alpha, s)$*

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

where function $seqLength(a)$ represents the length of a.

Intuitively, set $maxPoss(\alpha, s)$ is a collection of the maximal possible executable choices of α in the situation s. We prove the following intuitive property by using complete induction.

Theorem 2 *In the probabilistic domain specified properly satisfying conditions (a) and (b) in Section 2, for any stochastic sequential action $\alpha = \alpha_1; \cdots ; \alpha_n$, we have*

$$\vee(A = A_1; \cdots ; A_n \wedge choiceMac(\alpha, A) \wedge Poss(A, s)) \supset \sum_{A \in maxPoss(\alpha, s)} probMac(A, \alpha, s) = 1.$$

Up to now, everything works properly for sequential actions, and it is reasonable for us to consider that the *macro-action* can be of the form of sequential stochastic actions. To distinguish macro-actions from normal complex actions so that agents can recognize them, we introduce terms `macro` and `endmacro` such that

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

where $n \geq 2$ and $\alpha_i (1 \leq i \leq n)$ are stochastic actions, meaning that the sequential action $\alpha_1; \alpha_2; \cdots ; \alpha_n$ is treated as a macro-action named p_{name}. The definition of predicates introduced above for stochastic sequences can also be used for the name of macro-actions. In the sequel, we denote language $\mathcal{L}_{sc}$ with the notation extensions $do(a_1; \cdots ; a_n, s)$ and predicate $Poss(A_1; \cdots ; A_n, s)$ as language $\mathcal{L}'_{sc}$. Similar to Definition 1, the concept of s-regressable formulas can also be defined for formulas in $\mathcal{L}'_{sc}$.

3.2 The Knowledge Base for Macro-actions

The purpose of having a *knowledge base* for macro-actions is that an autonomous agent can reuse local information about 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 the environment at different times.

Suppose we have specified $\mathcal{D}$ and probabilities $prob_0$ for a stochastic system. We will develop a knowledge base for macro-actions introduced by controllers. As designed, the knowledge base is composed as follows:

[2]Similarly, we can also give an extended definition, called s-prime functional fluent, to the *prime* functional fluent [Reiter, 2001].

Definitions of macro-actions with their proper syntactic form. Moreover, a function $currentMaxLength$ is used to denote the maximal length of all the macro-actions in current knowledge base before new macro-actions are added. Initially, when the knowledge base is empty, we have the fact $currentMaxLength = 0$; when new macro-actions are added, the value n will be updated.

Extended axioms including the following three sub-groups:

a. The Extended Successor State Axioms for Fluents as discussed in Section 3.2 for every fluent in $\mathcal{D}$.

b. The Extended Precondition Axioms for nature's choices of macro-actions as discussed in Section 3.2. Since later we will implement the knowledge base in **Prolog**, by the closed world assumption (CWA) [Reiter, 1978], we need not keep those $Poss(A_1; \cdots ; A_m, s)$'s that are equivalent to $false$ after regression and simplification.

c. The Extended Probabilities. To achieve the goal of reusing useful results of macro-actions rather than recomputing them, we prefer to save the regression results for the definition of $probMac(A, \alpha, s)$.

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. These facts, $maxPossBase(List, \alpha, S)$, depend on particular situations, therefore are related to the initial database and programs. They will be generated during execution and will disappear when the controller reloads new initial database. Therefore, we call this the *dynamic* part, and its generation is embedded into the application interpreters. We call the former two components as the *static* part.

3.3 An Extended Regression Operator

To help us develop the knowledge base formally and later reuse the extended axioms in the base, a new regression operator $\mathcal{R}^\star$ is defined on s-regressable formulae in $\mathcal{L}'_{sc}$, where s is either S_0 or a variable of sort situation. Roughly speaking, $\mathcal{R}^\star$ is an extension of $\mathcal{R}$, and its aim is to try to use existing extended axioms first during regression, which therefore might save us computational steps.

Definition 3 (extended regression operator)
(1) *If $W = Poss(\alpha(\vec{t}), \sigma)$ where $\alpha(\vec{t})$ is a sequence of deterministic actions of length n and σ is of sort situation, and there is (extended) precondition axiom of the form $Poss(\alpha(\vec{x}), s_1) \equiv \Pi_\alpha(\vec{x}, s_1)$, then*[3] *$\mathcal{R}^\star[W] = \mathcal{R}^\star[\Pi_\alpha(\vec{t}, \sigma)]$; otherwise, we have*[3]

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

(2) *If W is an s-regressable atom, but not a $Poss$ atom, there are three possibilities:*
(a) s is the only situation-sort term (if any) mentioned by W, then $\mathcal{R}^\star[W] = W$.
(b) If 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 $m > 0$. If there is axiom of the 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, then[3]

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

otherwise, suppose $f(\vec{x}, do(a, s_1)) = y \equiv \phi_f(\vec{x}, y, a, s_1)$ is in $\mathcal{D}_{ss}$, then[3] *let $\sigma_1 = do(\alpha_1; \cdots ; \alpha_{m-1}, \sigma)$ 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 axiom 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, then[3] *$\mathcal{R}^\star[W] = \mathcal{R}^\star[\psi_F(\vec{t}, \alpha_1, \cdots , \alpha_n, \sigma)]$; otherwise, $F(\vec{x}, do(a, s_1)) \equiv \Phi_F(\vec{x}, a, s_1)$ is in $\mathcal{D}_{ss}$, then*[3] *$\mathcal{R}^\star[W] = \mathcal{R}^\star[\Phi_F(\vec{t}, \alpha_n, \sigma_1)]$, where $\sigma_1 = do(\alpha_1; \cdots ; \alpha_{n-1}, \sigma)$.*
(3) *For non-atomic formulas, regression is defined inductively as follows: $\mathcal{R}^\star[\neg W] = \neg\mathcal{R}^\star[W]$, $\mathcal{R}^\star[W_1 \wedge W_2] = \mathcal{R}^\star[W_1] \wedge \mathcal{R}^\star[W_2]$, and $\mathcal{R}^\star[(\exists x)W] = (\exists x)\mathcal{R}^\star[W]$.*

Because regression repeatedly substitutes logically equivalent formulas for atoms, what the operator delivers will be logically equivalent with what it starts with, i.e.,

Theorem 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 $\mathcal{R}^\star[W]$ 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 [Reiter, 2001], we also have the following properties:

Theorem 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*
(1) $\mathcal{D} \models \mathcal{R}[W] \equiv \mathcal{R}^\star[W]$;
(2) $\mathcal{D} \models W \; iff \; \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}^\star[W]$.

Moreover, notice that $do(a_1; \cdots ; a_n, s)$ represents the same situation as $do([a_1, \cdots , a_n], s)$ 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 $Poss(A, \sigma)$ in W_1 with equivalent

$$Poss(A_1, \sigma) \wedge \cdots \wedge Poss(A_n, do([A_1, \cdots , A_{n-1}], \sigma))$$

where $A = A_1; \cdots ; A_n$ and replacing any $do(a_1; \cdots ; a_n, \sigma)$ in W_1 with $do([a_1, \cdots , a_n], \sigma)$. We call W_2 as the *equal formula of W_1 in $\mathcal{L}_{sc}$*, and it is easy to see that

Theorem 5 *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*

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

[3]Without loss of generality, for any two first-order formulas in $\mathcal{L}'_{sc}$, we can always assume that all quantifiers (if any) of one formula have had their quantified variables renamed to be distinct from the free variables (if any) of the other.

These mean that our regression operator $\mathcal{R}^\star$ can perform more broadly, but still, it can achieve the same functions as the original operator $\mathcal{R}$.

By using $\mathcal{R}^\star$, we designed an recursive algorithm for agents so that the agents can develop or expand the static part of the knowledge base for macro-actions in advance by calling this program. The reason for using $\mathcal{R}^\star$ is obvious. For example, to compute the regression result of $F(\vec{x}, do(a_1; \cdots; a_n, s))(n > 1)$ given that all the extended successor state axioms for $a_1; \cdots; a_{n-1}$ have been computed and stored, we only need 2 steps by using $\mathcal{R}^\star$, while need n steps by using $\mathcal{R}$. Finally, we implemented this algorithm in **Prolog** as a program *kbDeveloper(list,KB)*, which is run by the agents when the controller demands to add the new macro-actions given in *list* into knowledge base *KB*. *KB* is a Prolog-program form file. We also performed successful experiments on a few stochastic systems including Example 1.

4 The Reuse of the Macro-actions

Our ultimate aim in defining macro-actions and developing knowledge base for them is to reuse the intermediate information. We specify an interpreter *macGolog* which takes programs composed of macro-actions and normal complex actions without changing the functions of stGolog interpreter.

Our new interpreter, *macDo*, expects a sequence $\alpha_1; \cdots; \alpha_n; nil$, 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, and is defined as follows:

$$macDo(nil, p, s, s') \overset{def}{=} s = s' \wedge p = 1;$$
$$macDo((\alpha; \beta); \gamma, p, s, s') \overset{def}{=} macDo(\alpha; (\beta; \gamma), p, s, s');$$

if α is a macro-action and we have had developed the static part of the knowledge base, then

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

where $shorter(a, b) \equiv seqLength(a) < seqLength(b)$; otherwise, if α is a stochastic action, $macDo$ works same as $stDo$:

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

Although we gave the definition of $maxPoss(\alpha, s)$, we still did not discuss how to practically generate it and keep it as a fact $maxPossBase(L, \alpha, s)$ which forms the dynamic part of the knowledge base. The following describes a simple way: if $maxPossBase(L, \alpha, s)$ is in the knowledge base, then checking $c \in maxPoss(\alpha, s)$ is same as checking $c \in L$; else, we will compute list V, L such that $V = \{A | choiceMac(\alpha, A) \wedge Poss(A, s)\}$ and $L = \{A | Poss(A, s) \wedge \neg(\exists A_1)[A_1 \in V \wedge Poss(A_1, s) \wedge realPrefix(A, A_1)\}$ in which $realPrefix(a, c)$ means a is a prefix of c and $a \neq c$, then assert the fact $maxPossBase(L, \alpha, s)$ into system, and now checking $c \in maxPoss(\alpha, s)$ is same as checking $c \in L$.

According to our definition of $macDo$, we have:

Lemma 1

1. *For any situation s and a sequence of stochastic actions α,*
   $macDo(\alpha; nil, p, s, s') \equiv stDo(\alpha; nil, p, s, s')$;
2. *for any situation s, 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), and γ be a sequence of combinations of stochastic actions and macro-actions followed by nil (including special case $\gamma = nil$), then*
   $macDo(\alpha; \beta; \gamma, p, s, s') \equiv macDo(\alpha; \Delta; \gamma, p, s, s')$.

Theorem 6 *For any situation s and any sequence $\alpha = \alpha_1; \cdots; \alpha_n$ where every α_i is either a stochastic action or a macro-action with body Δ_i, we have*

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

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.

Theorem 6 is proved by induction. It indicates that although we extend the interpreter with macro-actions, the function of the interpreter stays the same. So, what's the advantage of using macro-actions? It is all for the purpose of saving computational time, which will been seen later.

Other descriptions of $macDo$ for ?, *if-then-else* and *while* are same as $stDo$. We implemented the macGolog interpreter in **Prolog** and applied it successfully. Similar to stGolog, we also can define the probabilities $probF(\psi, \gamma)$ that some situation-suppressed sentence ψ will be true after executing a macGolog program γ simply by replacing $stDo$ to be $macDo$ in the Definition of $probF$ given in Section 2.

5 Computational Benefit – An Example

We discuss the advantage of introducing macro-actions by illustrating Example 1. Suppose the controller has given the description of the basic theory $\mathcal{D}$ for this system, and would like to consider the following procedure:

```
proc climbing(h)                                                  (P-1)
  ?(legalStair(h)); liftUpperLeg(h); forwLowLeg;
  stepDown(main); mvBaryct(main); straightLeg;
  forwSupLeg; stepDown(spleg); mvBaryct(spleg);
endproc
```

meaning after checking a stair of height h is capable of climbing, the robot performs a sequence of stochastic actions to climb the stair. In the above body of procedure (P-1), we can define the sequence from $liftUpperLeg(h)$ to $straightLeg$ as macro-action $stepMain(h)$ and the sequence from $forwSupLeg$ to $mvBaryct(spleg)$ as macro-action $stepSupp$. Then, (P-1) can be redefined to procedure (P-2) with body changed to $?(legalStair(h)); stepMain(h); stepSupp$. Similarly, we also can define the sequence from $liftUpperLeg(h)$ to $mvBaryct(spleg)$ as one macro-action $climbStair(h)$, and (P-1) is redefined as procedure (P-3) with body changed to $?(legalStair(h)); climbStair(h)$.

Compared to the time saved by reusing macro-actions, the time of developing the knowledge base (static part) for re-

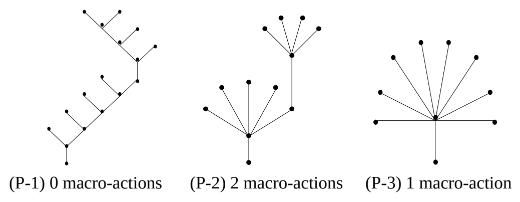


Figure 2: Computational Trees

stricted length macro-actions is fixed and negligible. Therefore, we will ignore the time consumed by developing the knowledge base and concentrate on the experiments of applications of using and reusing macro-actions.

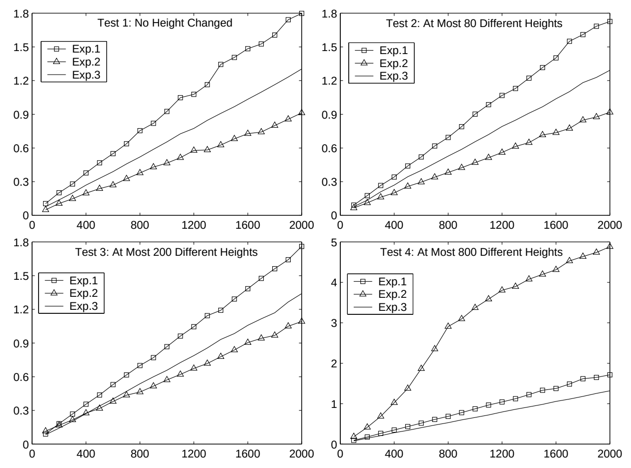
Provided that we will call these procedures (P-1), (P-2) and (P-3) repeatedly at the same local initial situation where the robot's status is ready in front of a stair shown in Figure 1, the outcome trees without or with macro-actions look like in Figure 2. Theoretically, regardless the time for searching knowledge base, the longer the macro-action is, the shorter the computational time should be. Because with macro-actions the computational trees are shorter, i.e., the computing steps are less. Ideally, it is nearly $\frac{t}{n}$ comparing to the time t consumed without macro-actions, where n is the maximal length of the macro-actions in the base.

But, in practice, the size of the knowledge base will affect the computational time, especially under our current preliminary implementation [Gu, 2003]. We did different tests for different cases of whether or not to use macro-actions over various environments, the stairs' heights change frequently. We considered three cases: (P-1) running under stGolog denoted as *Exp.1* and (P-2) (respectively, (P-3)) running under macGolog denoted as *Exp.2* (respectively, *Exp.3*). The knowledge bases respectively for Exp.2 and Exp.3 have been obtained by running $kbdeveloper$. We performed four tests for each case. For each test, the number of stairs N varies from 100 to 2000, and from *Test 1* to *Test 4* the change of the stairs' heights becomes more and more frequent. The CPU time with unit *second* is an average of ten distinct trials (to reduce the measurement error). The experimental results are shown in Figure 3.

Although searching the knowledge base takes time, we still can get computational benefit by choosing proper macro-actions under different environments. Moreover, we somehow make the agent have "memories", therefore can keep its "experience". The limitations, especially under our current implementation, are that, first, the duty of the controllers becomes heavier (it is the controller's responsibility to choose proper macro-actions); second, the agent may not be aware of similar local situations fully by itself, which causes unexpected database redundancy when we keep the dynamic part of the knowledge base. For example, in Exp.2, the agent did not aware that $do(stepMain(h), S_0)$ is also a class of similar local situations for all legal height h. We think this should not be difficult to fix later.

6 Conclusion and Future Work

In the context of stochastic systems, we introduced macro-actions based on the Golog sequential action constructor, extended the basic action theories, defined an extended regression operator to help us develop the knowledge base for macro-actions and reuse the saved information. Therefore, we partially simulate the behaviors that agents keep intermediate knowledge and later retrieve for computational benefit.



X-axes: number of stairs, Y-axes: CPU time

Figure 3: Experimental Results of Robot Climbing Stairs

Actually, we think macGolog (the modified stGolog) is not a very appropriate application of reusing macro-actions, since it needs to keep almost all the intermediate information of macro-actions, which did not fully show the benefit of reusing macro-actions. Notice that macro-action is more like a restricted type of local policy for some local Markov decision process (MDP). In the future, we want to loosen the structure of the macro-action, study its characters, optimize the structure of the knowledge base and make it more condensed. Moreover, we want to design a more proper interpreter for reusing macro-action in decision-making, say, maybe a modified dtGolog involving macro-action. In another word, our ultimate goal is high-level control, allowing agents to explicitly make optimal or nearly optimal choices more efficiently by reusing local optimal solutions of macro-actions in decision-theoretic planning.

References

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

[Boutilier *et al.*, 2000] Craig Boutilier, Raymond Reiter, Mikhail E. Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of Seventeenth National Conference on Artificial Intelligence*, pages 355–362, 2000.

[Greenwald, 1995] Lloyd Greenwald. How to avoid 'thinking on your feet'. In *Extending Theories of Action: Formal Theory and Practical Applications: Papers from the 1995 AAAI Spring Symposium*, pages 88–93. AAAI Press, Menlo Park, California, 1995.

[Gu, 2003] Yilan Gu. Handling uncertainty systems in the situation calculus with macro-actions. Master thesis, Dept. of Computer Science, University of Toronto, 2003. http://www.cs.toronto.edu/~yilan/thesis/ms1.pdf.

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

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

[Peña Castillo and Wrobel, 2002] Lourdes Peña Castillo and Stefan Wrobel. Macro-operators in multirelational learning: a search-space reduction technique. In T. Elomaa, H. Mannila, and H. Toivonen, editors, *ECML02*, volume 2430 of *LNAI*, pages 357–368, August 2002.

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

[Reiter, 1978] Raymond 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.

[Reiter, 1991] Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380, San Diego, 1991. Academic Press.

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