High-level Robot Programming and Program Execution.

Mikhail Soutchanski
School of Computer Science
Ryerson University
350 Victoria Street
Toronto, Ontario, M5B 2K3, Canada
mes@cs.toronto.edu

Abstract

We propose a logical framework for robot programming which allows the seamless integration of explicit agent programming with planning and with monitoring of a plan when it is executed in the real world. Specifically, the *Golog* model allows one to partially specify a control program in a high-level, logical language, and provides an interpreter that, given a logical axiomatization of a domain, will determine a plan. We extend the Golog model by introducing execution specific features that include run-time sensing and recovering from failures if they occur during execution of actions from the plan.

1 Introduction

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

These recent developments introduced not only a general, semantically clear approach to modeling dynamic systems in classical predicate logic, but also techniques for designing computationally efficient high-level controllers: by varying the degree of non-determinism in a Golog program and by using operators that bound the scope of search, the programmer can influence the efficiency of search for an executable control sequence. Golog is specifically targeted towards developing complex robotics software. Within robotics, the two major paradigms—planning and programming—have largely been pursued independently. Both approaches have their advantages (flexibility and generality in the planning paradigm, performance of programmed controllers) and scaling limitations (e.g., the computational complexity of planning approaches, task-specific design and conceptual complexity for programmers in the programming paradigm). Golog allows for the seamless integration of programming and planning. If the agent programmer has enough knowledge of a given domain to be able to specify some (but not necessarily all) of the structure and the details of a good (or possibly optimal) controller, then this knowledge can be used to overcome computational limitations associated with planning. Those aspects left unspecified will be filled in by the agent itself, but must satisfy any constraints imposed by the program.

The main contribution of research considered in this paper is accounting for uncertainty and unmodeled dynamics in an environment. Once a sequence of actions has been determined from a Golog program, is has to be executed in the real world. Because actions may have unexpected effects, the agent must sense the environment and compare the measured effects of actions with their effects expected according to the incomplete logical model of the environment. From the perspective of this research, Golog is a convenient language because at each step of computation, the logical statements about the current logical model can be easily evaluated; in other words, the logical model of the world is easily accessible and maintained.

There are several approaches to designing efficient and reliable controllers in Golog. According to one perspective on the environment where the control program is supposed to operate, there is a probabilistic model of the environment and interaction of the agent with the environment can be characterized as a Markov Decision Process (MDP). In this case, the planning task is formulated as a decision-theoretic planning and once the agent computes an optimal policy this policy provides all information required during the execution if outcomes of some actions are different from expected. According to another perspective, there is no probabilistic information about the environment where the agent is planning to act, and the agent is not capable or has no time for acquiring probabilities of different effects of its actions (this includes the case when estimation of those probabilities is impractical due to the nature of application domain). This setting is common in cases when the robot has to explore an unfamiliar environment only once or a few times: due to scarcity of interaction experiences, the probabilistic model cannot be estimated precisely and cannot be built in advance. In this case, the uncertainty and dynamics of the environment can be accounted only by observing the real outcomes of actions executed by the agent, by determining possible discrepancies between the observed outcomes and the effects expected according to the logical model of the world and then by recovering, if necessary, from the relevant discrepancies. To recover the agent computes an appropriate correction of the program that is being executed. A logical framework for execution monitoring of Golog programs (composed from deterministic primitive actions) provides the aforementioned functionalities and generalizes those previously known approaches to execution monitoring which have been formulated only for cases when the agent is given a linearly or partially ordered *sequence* of actions, but not an arbitrary *program*. Papers [De Giacomo *et al.*, 1998, Soutchanski, 1999] consider two general recovery mechanisms that can be employed by the execution monitor: a planning procedure and backtracking, respectively. The planning procedure computes off-line a short corrective sequence of actions such that after doing those actions on-line, the agent can resume executing the remaining part of the program. Backtracking to previous choice points in the non-deterministic Golog program gives an agent a chance to choose an alternative execution branch in cases when the further execution of actions remaining in the current branch is no longer possible (a post-condition of the program will not be satisfied) or desirable (there are other execution branches with higher utility). For example, this happens if the most recent action was executed overly late and no matter how fast the agent will execute actions remaining in the branch, it will miss a deadline at the end. In the general case, when the agent has executed actions after choosing a particular branch of a non-deterministic Golog program, the generalized recovery mechanism may use both planning and backtracking procedures if it is not possible or desirable to continue the execution of the current branch. The planning procedure is responsible in this case for computing a plan that leads back to a program state where an alternative branch can be chosen. This paper provides a declarative framework for this more general case when a recovery procedure can combine planning and backtracking into a single powerful recovery mechanism.

In Section 2 we review briefly the situation calculus, Golog and a transition-based semantics that specifies a Golog interpreter. In Section 3 we consider a declarative framework for monitoring of Golog programs. In Section 4 we introduce a specific implementation of a recovery procedure that generalizes previously proposed recovery procedures.

2 The Situation Calculus and Golog

2.1 The Situation Calculus

The situation calculus is a first-order language for axiomatizing dynamic worlds. In recent years, it has been considerably extended beyond the "classical" language to include concurrency, continuous time, etc., but in all cases, its basic ingredients consist of *actions*, *situations* and *fluents*.

Actions

Actions are first-order terms consisting of an action function symbol and its arguments. In the approach to representing time in the situation calculus of [Reiter, 2001], one of the arguments to such an action function symbol—typically, its last argument—is the time of the action's occurrence. For example, $startGo(l, l', 3.1)$ might denote the action of a robot starting to move from location l to l' at time 3.1. Following Reiter [Reiter, 2001], all actions are instantaneous (i.e, with zero duration).[1]

[1]Durations can be captured using processes, as shown below. A full exposition of time is not possible here.

Situations

A *situation* is a first-order term denoting a sequence of actions. These sequences are represented using a binary function symbol *do*: $do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. So $do(\alpha, s)$ is like LISP's $cons(\alpha, s)$, or Prolog's $[\alpha \mid s]$. The special constant S_0 denotes the *initial situation*, namely the empty action sequence. so S_0 is like LISP's ( ) or Prolog's [ ]. Therefore, the situation term

$$do(endGo(l, l', 7.3), do(startGrasp(o, 2), do(startGo(l, l', 2), S_0)))$$

denotes the following sequence of actions: $startGo(l, l', 2)$, $startGrasp(o, 2)$, $endGo(l, l', 7.3)$. Foundational axioms for situations without time are given in [Reiter, 2001]. Axioms for situations with time are given in [Reiter, 1998].

Fluents

Relations or functions whose truth values vary from state to state are called *relational fluents*, and are denoted by predicate or function symbols whose last argument is a situation term. For example, $closeTo(x, y, s)$ might be a relational fluent, meaning that when the robot performs the action sequence denoted by the situation term s, x will be close to y. $pos(x, s)$ might be a functional fluent, denoting x's position in that state of the world reached by performing the action sequence s.

A domain theory is axiomatized in the situation calculus with four classes of axioms:

Action precondition axioms: There is one for each action function $A(\vec{x})$, with syntactic form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$. Here, $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$. These are the preconditions of action A.

Successor state axioms: There is one for each relational fluent $F(\vec{x}, s)$, with syntactic 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 $a, s, \vec{x}$. These characterize the truth values of the fluent F in the next situation $do(a, s)$ in terms of the current situation s, and they embody a solution to the frame problem for deterministic actions ([Reiter, 1991]). There are similar axioms for functional fluents.

Unique names axioms for actions: These state that the actions of the domain are pairwise unequal.

Initial database: This is a set of sentences whose only situation term is S_0; it specifies the initial problem state.

Example 2.1: The following are action precondition and successor state axioms for a blocks world. To keep the example short, we suppose that blocks may only be moved onto other blocks. The axioms appeal to a process fluent $moving(x, y, t, t', s)$, meaning that block x is in the process of moving to y, and t and t' are the initiation and termination times of this process. The process has its own instantaneous initiating and terminating actions, $startMove(x, y, t)$ and $endMove(x, y, t)$, with the obvious meanings.

Action Precondition Axioms

$$Poss(startMove(x, y, t), s) \equiv clear(x, s) \wedge clear(y, s) \wedge x \neq y \wedge t = start(s),$$

$$Poss(endMove(x, y, t), s) \equiv (\exists t')moving(x, y, t', t, s).$$

Successor State Axioms

$$clear(x, do(a, s)) \equiv (\exists y, z, t)\{on(y, x, s) \wedge a = startMove(y, z, t)\} \vee clear(x, s) \wedge \neg(\exists y, t)a = endMove(y, x, t),$$

$$on(x, y, do(a, s)) \equiv (\exists t)a = endMove(x, y, t) \vee on(x, y, s) \wedge \neg(\exists z, t)a = startMove(x, z, t),$$

$$onTable(x, do(a, s)) \equiv onTable(x, s) \wedge \neg(\exists y, t) a = startMove(x, y, t).$$

$$moving(x, y, t, t', do(a, s)) \equiv a = startMove(x, y, t) \wedge t' = t + moveDuration(x, y, s) \vee moving(x, y, t, t', s) \wedge a \neq endMove(x, y, t').$$

2.2 Golog

Golog [Levesque *et al.*, 1997] is a situation calculus-based programming language for defining complex actions in terms of a set of primitive actions axiomatized in the situation calculus as described above. It has the standard—and some not-so-standard—control structures found in most Algol-like languages.

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

The semantics of Golog programs is defined by macro-expansion, using a ternary relation *Do*. $Do(\delta, s, s')$ is an *abbreviation* for a situation calculus formula whose intuitive meaning is that s' is one of the situations reached by evaluating the program δ beginning in situation s. Given a program δ, one *proves*, using the situation calculus axiomatization of the background domain, the formula $(\exists s)Do(\delta, S_0, s)$ to compute a plan. Any binding for s obtained by a constructive proof of this sentence is a sequence of possible actions, involving only primitive actions, of δ. A Golog interpreter for the situation calculus with time, implemented in Prolog, is described in [Reiter, 1998].

Example 2.2: The following is a nondeterministic Golog program for the example above. $makeOneTower(z)$ creates a single tower of blocks, using as a base the tower whose top block is initially z.

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

Like any Golog program, this is executed by proving

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

in our case, using background axioms above. We start with S_0 as the current situation. In general, if σ is the current situation, $makeOneTower(z)$ terminates in situation σ if $\neg(\exists y).y \neq z \wedge clear(y, \sigma)$ holds. Otherwise it nondeterministically selects a block x and time t, and "performs" $startMove(x, z, t)$, meaning it makes $do(startMove(x, z, t), \sigma)$ the current situation; then it picks a time t' and "performs" $endMove(x, z, t')$, making $do(endMove(x, z, t'), do(startMove(x, z, t), \sigma))$ the current situation; then it calls itself recursively. On termination, the current situation is returned as a side effect of the computation; this corresponds to a sequence of possible actions specified by the program. It is important to understand that this is an offline computation; the resulting situation is intended to be passed to some execution module for the online execution of the primitive actions mentioned in the situation term, in our example, to physically build the tower.

Thus the interpreter will makes choices (if possible) that lead to successful computation of one of the plans specified by the program. With nondeterministic choices and the specification of postconditions corresponding to goals, Golog can be viewed as integrating planning and programming in a natural way.

2.3 Transition-based semantics

This semantics is defined using axioms for *Trans* and *Final* that are discussed in [De Giacomo *et al.*, 2000]. The predicate $Trans(\delta_1, s_1, \delta_2, s_2)$ holds if a Golog program δ_1 makes a transition in situation s_1 to a structurally simpler program δ_2 and results in situation s_2. Given a program δ and a situation s, $Trans(\delta, s, \delta', s')$ tells us which is a possible next step in the computation, returning the resulting situation s' and the program δ' that remains to be executed. In other words, $Trans(\delta, s, \delta', s')$ denotes a transition relation between configurations (δ, s) and (δ', s'). The predicate $Final(\delta, s)$ holds if term δ is a program in a final (correctly terminated) state in situation s, that is whether the computation is completed (no program remains to be executed). The set of axioms provided in [De Giacomo *et al.*, 2000] characterizes these two predicates by induction on the structure of the program.

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

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

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

$T(\delta, s, \delta, s)$
$Trans(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \supset T(\delta, s, \delta', s')$

Using $Trans^*$ and *Final*, a definition of the $Do(\delta, s, s')$ relation is the following:

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

A Golog interpreter (that is correct with respect to a specification provided by *Trans* and *Final* axioms) has been implemented in Prolog (see [De Giacomo *et al.*, 2000] for details).

3 Execution Monitoring: The General Framework

In this section we elaborate on a framework developed in [De Giacomo *et al.*, 1998, Soutchanski, 1999], without committing to any particular details of the monitor: we introduce the notion of trace and expand the framework accordingly.

Once a sequence of actions is computed by a Golog interpreter, it cannot be simply given to an execution module that has to execute each action in the real world. Even in benign and well-structured office-type environments there is no guarantee that during the execution of actions truth values of relational fluents or values of functional fluents will remain in exact correspondence to their values in the real world: other agents perform their actions independently from the robot and those actions may change the actual values of fluents. The behavior of other agents is unpredictable and often cannot be modeled an advance[2], but the feedback from the external world can help to account for unpredictable effects and obtain the desirable closed-loop behavior. Thus, imagine that at any stage of planning, when the interpreter selects a primitive action for execution or a test condition for evaluation, a high-level control module of the robot executes a set of predefined sense actions[3] to compare reality with its internal representation (i.e., with values of fluents according to successor state axioms). If the robot does not notice any *relevant* discrepancies, then it executes the action in reality (or evaluates the test, respectively). Otherwise, the high-level control module (called the *monitor*) attempts to *recover* from unexpected discrepancies by finding a *corrected* program from the remaining part of the program and then proceeds with the corrected program or fails.[4] The processes of interpreting and execution monitoring continues until the program reaches the final configuration or fails. It is convenient to imagine that all discrepancies between the robot's mental world and reality are the result of sensory input, but our formal model remains the same if we assume that the high level control module is explicitly told also about exogenous actions.

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

$$trace(\delta_1, h_1) = trace(\delta_2, h_2) \supset \delta_1 = \delta_2 \wedge h_1 = h_2, \quad (1)$$

$$(\forall P).P(H_0) \wedge (\forall \delta, h)[P(h) \supset P(trace(\delta, h))] \supset (\forall h)\, P(h), \quad (2)$$

$$\neg(\exists \delta, h)\, trace(\delta, h) \prec H_0, \quad (3)$$

$$h \prec trace(\delta, h') \equiv h \preceq h'. \quad (4)$$

The axiom (2) is a second order induction axiom saying that the sort *trace* is the smallest set containing H_0 that is closed under the application of $trace$ to a *program* term and *trace*. The axiom (1) is a unique names axiom for traces that together with (2) imply that if two traces are the same, then each of them is the result of the same sequence of program states going from H_0 (informally, different sequences of program states forking from H_0 cannot join later resulting in the same trace). The axiom (3) means that H_0 has no predecessors, and (4) asserts that a trace h is a predecessor of other trace that results from adding a program state δ to a trace h' if and only if h is a predecessor of h' or h is equal to h'. These axioms are domain independent (notice that they are similar to foundational axioms for situations).

The closed-loop system (interpreter and execution monitor) is characterized formally by a new predicate symbol $TransEM(\delta_1, s_1, h_1, \delta_2, s_2, h_2)$, describing a one-step transition consisting of a single $Trans(\delta_1, s_1, \delta', s')$ step of program interpretation, followed by a process, called $Monitor$, of execution monitoring. The role of the execution monitor is to get new sensory input in the form of sense and exogenous actions and (if necessary) to generate a program to counter-balance any perceived discrepancy. As a result of all this, the system passes from (δ_1, s_1, h_1) to the configuration (δ_2, s_2, h_2) specified as follows:

$$\begin{array}{l} TransEM(\delta_1, s_1, h_1, \delta_2, s_2, h_2) \equiv \\ \quad (\exists\, \delta', s', h')Trans(\delta_1, s_1, \delta', s') \;\wedge h' = trace(\delta_1, h_1) \wedge \\ \quad (\exists s_g)\, Do(\delta', s', s_g) \wedge \exists s_n Monitor(s_1, \delta', s', h', \delta_2, s_n, h_2) \wedge \\ \qquad (s' = s_1 \wedge s_2 = s_n \vee \exists a.s' = do(a, s_1) \wedge \mathit{senseEffect}(a, s_n, s_2)), \end{array} \quad (5)$$

where the predicate *senseEffect*(a, s_n, s_2) holds if the situation s_2 results from executing a sequence of pre-specified sense actions in s_n. These sense actions are domain specific and we assume that a domain axiomatizer provides a sequence of sense actions that has to be executed after every primitive action a to find real effects of a. Note that in the axiom (5) either $\exists a.s' = do(a, s_1)$ (a first primitive action in δ_1 is selected for execution), or $s' = s_1$ (a test is evaluated). [5]

The possible configurations that can be reached with execution monitoring by a program δ given s and a trace h are those obtained by repeatedly following *TransEM* transitions, i.e. those in its reflexive transitive closure *TransEM**: *DoEM*$(\delta, s, h, s_f) \equiv \exists \delta_f, h_f.$*TransEM*$^*(\delta, s, h, \delta_f, s_f, h_f) \wedge$ *Final*(δ_f, s_f), where $TransEM^*$ is defined as the following second-order situation calculus formula:

$$(\forall \delta, s, h, \delta', s', h')TransEM^*(\delta, s, h, \delta', s', h') \stackrel{\text{def}}{=} \forall U[\ldots \supset U(\delta, s, h, \delta', s', h')]$$

and the ellipsis stands for the conjunction of :

$$U(\delta, s, h, \delta, s, h)$$

$$U(\delta, s, h, \delta', s', h') \wedge \mathit{TransEM}(\delta', s', h', \delta'', s'', h'') \supset U(\delta, s, h, \delta'', s'', h'')$$

Let e be an exogenous event, which might be as simple as a primitive sense action (e.g., sensing of the current time) or as complex as an arbitrary Golog program (e.g., that another agent is planning to execute). Then, $Do(e, s, s_e)$ holds if s_e is a situation resulting after doing e in s; informally, s_e takes into account an influence of the external real world. The behavior of a generic monitor is specified by:

$$\begin{array}{l} Monitor(s, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \equiv \exists e, s_e, a.Do(e, s, s_e) \wedge \\ \quad [\neg Relevant(s, s_e, \delta_1, s_1, h_1) \wedge \delta_2 = \delta_1 \wedge h_2 = h_1 \wedge \\ \qquad (s_1 = s \wedge s_2 = s_e \vee s_1 = do(a, s) \wedge s_2 = do(a, s_e)\,) \vee \\ \quad Relevant(s, s_e, \delta_1, s_1, h_1) \wedge s_2 = s_e \wedge \\ \qquad (\, s_1 = s \wedge Recover(s, \delta_1, h_1, s_e, \delta_2, h_2) \vee \\ \qquad\quad s_1 = do(a, s) \wedge Recover(s, (a; \delta_1), h_1, s_e, \delta_2, h_2)\,)]. \end{array} \quad (6)$$

Here, $Relevant(s, s_e, \delta_1, s_1, h_1)$ is a predicate that specifies whether the discrepancy between s and s_e is relevant in the current configuration of the program. If this discrepancy does not matter – $\neg Relevant(s, s_e, \delta_1, s_1, h_1)$ – then the execution monitor does not modify the program – $\delta_2 = \delta_1$, and

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

[3] All sense actions $sense(q, v, t)$ have an argument v which represents data returned from sensors at the run time t when the robot measures q.

[4] If the remainder of a program is long enough, then it seems computationally advantageous to recover from discrepancies and then use the remaining program rather than throw it away and plan from scratch.

[5] *Trans*$(\phi?, s, nil, s)$ if ϕ holds in s, i.e., transitions over tests do not change the situation argument; see [De Giacomo *et al.*, 2000] for details.

keeps the current trace: $h_2 = h_1$. In addition, if the last transition was over the test ($s_1 = s$), the monitor does nothing else, but if it was over a primitive action ($s_1 = do(a, s)$), the monitor does the action a in the real world (this execution results in a new situation $s_2 = do(a, s_e)$). Otherwise, if this discrepancy does matter – $Relevant(s, s_e, \delta_1, s_1, h_1)$ – the monitor should recover. The predicate $Recover(s, \delta, h_1, s_e, \delta_2, h_2)$ provides for this by determining (possibly, using h_1 and δ – which is either δ_1 or $a; \delta_1$) a new program, δ_2, whose execution in situation s_e is intended to achieve an outcome *equivalent* (in a sense left open for the moment) to that of program δ, had the exogenous event not occurred and δ was simply executed in s as it was originally expected.

The wide range of monitors can be achieved by defining $Relevant$ and $Recover$ in different ways. For example, corrective plans are suggested as a domain-independent recovery technique in [De Giacomo *et al.*, 1998]. Another example is when a new program δ_2 is determined by going back to a previous program state (which is a nondeterministic operator) and choosing an alternative branch; δ_2 can be found because the trace h_1 keeps all program states which the interpreter has visited already [Soutchanski, 1999].

It can be advantageous to combine in a recovery procedure two techniques: planning and backtracking to a previous computation state. Indeed, one can easily imagine the following recovery procedure. Try all length one sequences of actions (followed by a remaining program δ), if none succeeds, go back to a previous computation state and try from there all length one sequences of actions (again followed by a remaining program δ), if this does not work, go to an earlier computation state and search for a single corrective action from there, etc. If all these attempts fail, try all length two sequences of actions (followed by a remaining program δ), if none succeeds, go back to a previous computation state and try from there all length two sequences of actions, etc. This modified procedure will insert as few corrective actions as possible: it attempts to trade off corrective actions for backtracking to a previous program counter δ from which one can resume execution to reach a goal.[6] It is not difficult to see that this new recovery procedure is more general than the recovery procedure based solely on re-planning.

4 A Specific Monitor

Now we develop a simple realization of the above general framework, by fixing on particular predicates $Relevant$ and $Recover$.

We begin by assuming that for each application domain a programmer provides:

1. The specification of all primitive actions (robot's and exogenous) and their effects, together with an axiomatization of the initial situation, as described in Section 2.
2. A Golog program γ that may or may not take into account exogenous actions occurring when the robot executes the program. Before we give a suitable definition of $Recover$, we make an important assumption about the syntactic form of the monitored program γ. Specifically, we assume that along with her program, the programmer provides a first order sentence describing the program's goal, or what programmers call a program *postcondition*. We assume further that this postcondition is postfixed to the program. In other words, if γ is the original program, and $goal$ is its postcondition, then the program we shall be dealing with in this section will be $\gamma\, ; goal?$.

[6]Because certain primitive actions have been already executed in the real world, it is not always possible simply backtrack to a previous program counter and use it as a new program when we find that the execution of the current program fails in the situation that results from exogenous actions. It may be necessary to do one or several auxiliary actions, before computation can be resumed from one of the previous program counters.

Next, we take $Relevant(s, s_e, \delta_1, s_1, h_1)$ to be as follows:

$$\begin{array}{l} Relevant(s, s_e, \delta_1, s_1, h_1) \equiv \\ \quad (s_1 = s \supset \neg\exists s_g Do(\delta_1, s_e, s_g)\,)\wedge \\ \quad (\forall a).(s_1 = do(a, s) \supset \neg\exists s_g Do(a; \delta_1, s_e, s_g)\,), \end{array} \qquad (7)$$

so that the definition (6) of $Monitor$ becomes:

$$\begin{array}{l} Monitor(s, \delta_1, s_1, h_1, \delta_2, s_2, h_2) \equiv \exists e, s_e, a. Do(e, s, s_e)\wedge \\ \quad [\,(s_1 = s \wedge \exists s_g Do(\delta_1, s_e, s_g) \wedge \delta_2 = \delta_1 \wedge s_2 = s_e \wedge h_2 = h_1 \vee \\ \quad\; s_1 = do(a, s) \wedge \exists s_g Do(a; \delta_1, s_e, s_g)\wedge \\ \qquad\qquad \delta_2 = \delta_1 \wedge s_2 = do(a, s_e) \wedge h_2 = h_1\,) \vee \\ \quad (s_1 = s \supset \neg\exists s_g Do(\delta_1, s_e, s_g)\,)\wedge \\ \qquad (\forall a).(s_1 = do(a, s) \supset \neg\exists s_g Do(a; \delta_1, s_e, s_g)\,) \wedge s_2 = s_e \wedge \\ \qquad (\,s_1 = s \wedge Recover(s, \delta_1, h_1, s_e, \delta_2, h_2) \vee \\ \qquad\; s_1 = do(a, s) \wedge Recover(s, (a; \delta_1), h_1, s_e, \delta_2, h_2)\,)]. \end{array}$$

$Monitor$ checks for the existence of an exogenous program e, determines the situation s_e reached by this program, and if the monitored program δ_1 terminates off-line, the monitor returns δ_1, else it invokes a recovery mechanism to determine a new program δ_2. Therefore, $Monitor$ appeals to $Recover$ only as a last resort; it prefers to let the monitored program keep control, so long as this is guaranteed to terminate off-line in a situation where the program's goal holds. (Remember that this goal has been postfixed to the original program, as described above.)

It only remains to specify the predicate $Recover(s, \delta, h_1, s_e, \delta_2, h_2)$ that is true whenever δ is the current state of the program being monitored[7], s is the situation prior to the occurrence of the exogenous program, s_e is the situation after the exogenous event, h_1 is the current trace, and δ_2 is a new program to be executed on-line in place of δ, beginning in situation s_e. Notice that here we are appealing to the assumption 2 above that all monitored programs are postfixed with their goal conditions. We need something like this because the recovery mechanism *changes* the program δ being monitored, by adding a prefix p to it. The resulting program $(p; \delta)$ may well terminate, but in doing so, it may behave in ways unintended by the programmer. But so long as the goal condition has been postfixed to the original program, all terminating executions of the altered program will still satisfy the programmer's intentions.

$$\begin{array}{l} Recover(s, \delta, h_1, s_e, \delta_2, h_2) \equiv \exists p. straightLineProg(p)\wedge \\ \quad (\,\exists s_g. Do(p\,; \delta, s_e, s_g) \wedge \delta_2 = (p\,; \delta) \wedge h_2 = h_1 \vee \\ \quad \neg\exists s_g. Do(p\,; \delta, s_e, s_g) \wedge \exists(\delta', h'). firstAlternative(h_1, \delta', h')\wedge \\ \qquad\qquad\qquad\qquad Recover(s, (p; \delta'), h', s_e, \delta_2, h_2)\,)\wedge \\ \quad [\forall(p', s', \delta_2', h_2'). straightLineProg(p')\wedge \\ \qquad (\,Do(p'\,; \delta, s_e, s') \vee \exists(\delta', h'). firstAlternative(h_1, \delta', h')\wedge \\ \qquad\qquad Recover(s, (p'; \delta'), h', s_e, \delta_2', h_2')\,) \supset \\ \qquad\qquad\qquad length(p) \leq length(p')\,], \end{array} \qquad (8)$$

[7]If the last transition was over the test, then δ is δ_1, but if the last transition was over a primitive action a, then δ is $(a; \delta_1)$, where δ_1 is the current program counter.

where the predicate $firstAlternative(h_1, \delta', h')$ holds if δ' is a first (if one looks backwards from the current trace h_1 towards H_0) program state mentioned in the trace h_1 such that δ' is a nondeterministic choice between several sub-programs.

This predicate is specified by

$$
\begin{array}{l}
firstAlternative(h_1, \delta', h') \equiv \\
\quad h' \preceq h_1 \wedge \exists(h'') h' = trace(\delta', h'') \wedge \\
\qquad\qquad \exists(\gamma_1, \gamma_2, \gamma_3).(\delta' = (\gamma_1 \mid \gamma_2) \vee \delta' = \pi x.\gamma_3) \wedge \\
\quad \forall(h).(h \prec h_1 \wedge \exists(h'', \delta, \gamma_1, \gamma_2, \gamma_3).h = trace(\delta, h'') \wedge \\
\qquad\qquad (\delta = (\gamma_1 \mid \gamma_2) \vee \delta = \pi x.\gamma_3) \supset h \preceq h').
\end{array} \tag{9}
$$

Informally, given a trace h_1, this predicate computes the sub-trace h' of h_1 such that this sub-trace is composed by a program state δ' that is a nondeterministic choice between sub-programs and h' is the longest sub-trace that has this property. In other words, all other sub-traces h that are composed from program states which are nondeterministic choices between sub-programs are further away in the past than the sub-trace h'.

Note that our new recovery procedure (8) will determine a shortest (possibly empty) sequence of actions such that execution of actions (if any) from this sequence followed by current program state or followed by one of the past program states leads to a situation where the program postcondition holds. Moreover, only those past program states are considered which are nondeterministic choices between several sub-programs. In [Soutchanski, 1999], we consider a special but interesting case of this general recovery mechanism: when no corrective actions are inserted in front of the past program states and a new program state is determined by the recovery procedure exclusively from backtracking to a past program state.

Formally, the predicate $straightLineProg(p)$ and the function $length(p)$ in (8) are defined as follows:

$$
\begin{array}{l}
straightLineProg(q) \equiv (\forall P, a).[\, P(?(true)) \wedge \\
\quad (straightLineProg(q) \wedge primitive\_action(a) \supset P(a;q)) \\
\qquad\qquad \supset P(q)\,]. \\
length(?(true)) = 0 \wedge length(a) = 1 \wedge \\
\quad straightLineProg(p) \supset length(a;p) = 1 + length(p).
\end{array}
$$

This recovery procedure has a built-in assumption that all exogenous events, if not neutral with respect to achieving the goal, are malicious.

5 Discussion

We would like to make here a few comments regarding our choice of the recovery procedure. First, there are other alternative specifications, e.g., if an exogenous disturbance happens when the remainder of the program is short, it can be more advantageous to find a sequence of actions that leads directly to the goal, rather than recover from the disturbance and resume execution of the remaining program. We do not provide here alternative specifications because our intention is to introduce a conceptual framework; more sophisticated versions of recovery mechanism will be minor variations of the recovery mechanism considered here. Second, the suggested implementation of recovery procedure acceptable only if the upper bound on the length of p is a small number. For any given application domain, computational efficiency of the implementation can be significantly improved by introducing domain specific declarative constraints on search for an appropriate $straightLineProg(p)$. It is well known that this approach leads to efficient domain specific planning [Kibler and Morris, 1981, Bacchus and Kabanza, 1996, Kvarnström and Doherty, 2001]. Note that an efficient implementation of $straightLineProg(p)$ based on declarative constraints on the search for a shortest sequence will be applicable to the task of monitoring of an arbitrary program in the domain of application. The recovery procedure will remain the same as long as programs use primitive actions from the same basic theory of actions. Third, in the real world, if the initial situation is different from what programmer has expected, it may happen that the original program $Prog$ cannot be started. However, in this case we can simply run monitor once, find what kind of disturbance occurred, obtain a new (corrected) version of the program $ProgCorr$ and start executing this program. Forth, a programmer might wish to provide several intermediate postconditions in different parts of a program: these postconditions set expectations of what those parts of the program have to achieve notwithstanding exogenous disturbances.

A more detailed discussion of previously proposed approaches to execution monitoring can be found in papers [De Giacomo *et al.*, 1998, Soutchanski, 1999]. There are several systems designed to interleave monitoring with plan execution: PLANEX [Fikes *et al.*, 1972], IPEM [Ambrose-Ingerson and Steel, 1988], SIPE [Wilkins, 1988], ROGUE [Haigh and Veloso, 1996], SOAR [Rosenbloom *et al.*, 1993] and others. We differ from these and similar proposals, first by using the very expressive predicate logic language for specifying application domains, secondly by the fact that ours is a story for monitoring arbitrary *programs*, not simply straight line or partially ordered plans. Moreover, we do not assume that the monitored plan is generated automatically from scratch, but rather that it has been computed from a high-level Golog program provided by a programmer.

6 Future Work

This paper is intended to propose a general predicate logic based framework to integration of high-level programming with execution. However, this framework needs a detailed experimental evaluation and computational analysis of its applicability to different robotics scenarios. These remain important topics for further research.

It would be interesting to develop a logic-based execution monitoring framework that accounts for time required by an interpreter to determine a plan and for time that recovery procedure takes to find a corrected program.

The robotics implementation considered in [Soutchanski, 1999] demonstrated that the logical framework for monitoring of high-level programs facilitates development of interesting robotics implementations. An implementation of a more general approach considered in this paper is one of the possible research directions.

References

[Ambrose-Ingerson and Steel, 1988] J. Ambrose-Ingerson and S Steel. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 735–740. Morgan Kaufmann, 1988.

[Bacchus and Kabanza, 1996] Fahiem Bacchus and Froduald Kabanza. Using temporal logic to control search in a forward chaining planner. In M. Ghallab and A. Milani, editors, *New Directions in Planning*, pages 141–153, Amsterdam, 1996. IOS Press.

[De Giacomo *et al.*, 1998] G. De Giacomo, R. Reiter, and M.E. Soutchanski. Execution monitoring of high-level robot programs. In *Principles of Knowledge Representation and Reason-*

ing: Proc. of the 6th International Conference (KR'98), pages 453–464, Trento, Italy, 1998.

[De Giacomo *et al.*, 2000] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[Fikes *et al.*, 1972] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Learning and executing generalized robot plans. Artificial Intelligence, 3(4):251–288, 1972.

[Haigh and Veloso, 1996] K.Z. Haigh and M. Veloso. Interleaving planning and robot execution for asynchronous user requests. In *International Conf. on Intelligent Robots and Systems (IROS-96)*, pages 148–155, Osaka, Japan, 1996.

[Kibler and Morris, 1981] D. Kibler and P. Morris. Do not be stupid. In *Proc. of the 7th International Joint Conference on Artificial Intelligence, IJCAI-1981*, pages 345–347, Vancouver, BC, Canada, 1981.

[Kvarnström and Doherty, 2001] J. Kvarnström and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2001.

[Levesque *et al.*, 1997] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog : A logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31, N 1–3:59–83, 1997.

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

[Reiter, 1998] R. Reiter. Sequential, temporal golog. In A.G. Cohn and L.K. Schubert, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the 6th International Conference (KR'98)*, pages 547–556, Trento, Italy, 1998.

[Reiter, 2001] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, 2001.

[Rosenbloom *et al.*, 1993] P.S. Rosenbloom, J.E. Laird, and A. Newell. *The Soar Papers: Readings on Integrated Intelligence*. MIT Press, Cambridge, MA, 1993.

[Soutchanski, 1999] Mikhail Soutchanski. Execution monitoring of high–level temporal programs. In Michael Beetz and Joachim Hertzberg, editors, *Robot Action Planning, Proceedings of the IJCAI-99 Workshop*, pages 47–54, Stockholm, Sweden. Available `http://www.cs.toronto.edu/~ mes/papers`, 1999.

[Wilkins, 1988] D.E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers, San Francisco, CA, 1988.