Execution Monitoring of High-Level Robot Programs.

Giuseppe De Giacomo[*]
Dip. di Informatica e Sistemistica
Università di Roma "La Sapienza"
Roma, I-00198, Italy
degiacomo@dis.uniroma1.it

Ray Reiter
Dept. of Computer Science
University of Toronto
Toronto, M5S 3G4, Canada
reiter@cs.toronto.edu

Mikhail Soutchanski
Dept. of Computer Science
University of Toronto
Toronto, M5S 3G4, Canada
mes@cs.toronto.edu

Abstract

Imagine a robot that is executing a program on-line, and, insofar as it is reasonable to do so, it wishes to continue with this on-line program execution, no matter what exogenous events occur in the world. *Execution monitoring* is the robot's process of observing the world for discrepancies between the actual world and its internal representation of it, and recovering from such discrepancies.

We provide a situation calculus-based account of such on-line program executions, with monitoring. This account relies on a specification for a single-step interpreter for the logic programming language *Golog* . The theory is supported by an implementation that is illustrated by a standard blocks world in which a robot is executing a *Golog* program to build a suitable tower. The monitor makes use of a simple kind of planner for recovering from malicious exogenous actions performed by another agent. After performing the sequence of actions generated by the recovery procedure, the robot eliminates the discrepancy and resumes executing its tower-building program.

We also indicate how, within the formalism, one can formulate various correctness properties for monitored systems.

1 Introduction and motivation.

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

1. It can give up trying to complete the execution of its program.
2. It can call on its programmer to give it a more sophisticated program, one that anticipates all possible discrepancies between the actual world and its internal model, and that additionally instructs it what to do to recover from such failures.
3. It can have available to it a repertoire of *general* failure recovery methods, and invoke these as needed. One such recovery technique involves planning; whenever it detects a discrepancy, the robot computes a plan that, when executed, will restore the state of the world to what it would have been had the exogenous action not occurred. Then it executes the plan, after which it resumes execution of its program.

Execution monitoring is the robot's process of observing the world for discrepancies between "physical reality", and its "mental reality", and recovering from such perceived discrepancies. The approach to execution monitoring that we take in this paper is option 3 above. While option 2 certainly is valuable and important, we believe that it will be difficult to write programs that take into account all possible exceptional cases. It will be easier (especially for inexperienced programmers) to write simple programs in a language like *Golog* , and have a sophisticated execution monitor (written by a different, presumably more experienced programmer) keep the robot on track in its actual execution of its program.

In general, we have the following picture: The robot is executing a program on-line. By this, we mean that it is physically performing actions in sequence, as these

[*]Author names are alphabetical.

are specified by the program.[1] After each execution of a primitive action or of a program test action, the execution monitor observes whether an exogenous action has occurred. If so, the monitor determines whether the exogenous action can affect the successful outcome of its on-line execution. If not, it simply continues with this execution. Otherwise, there is a serious discrepancy between what the robot sensed and its internal world model. Because this discrepancy will interfere with the further execution of the robot's program, the monitor needs to determine corrective action in the form of another program that the robot should continue executing on-line instead of its original program. So we will understand an execution monitor as a mechanism that gets output from sensors, compares sensor measurements with its internal model and, if necessary, produces a new program whose on-line execution will make things right again.

Our purpose in this paper is to provide a situation calculus-based account of such on-line program executions, with monitoring. To illustrate the theory and implementation, we consider a standard blocks world as an environment in which a robot is executing a *Golog* program to build a suitable tower. The monitor makes use of a simple kind of planner for recovering from malicious exogenous actions performed by another agent. After the robot performs the sequence of actions generated by the recovery procedure, the discrepancy is eliminated and the robot can resume building its goal tower.

2 The Situation Calculus and Golog

The version of the situation calculus that we use here has been described in [18], [22], and elsewhere. The situation calculus is a second order language specifically designed for representing dynamically changing worlds. All changes to the world are the result of named *actions*. A possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant S_0 is used to denote the *initial situation*, namely the empty history. Non-empty histories are constructed using a distinguished binary function symbol do; $do(\alpha, s)$ denotes the successor situation to s resulting from performing the action α. Actions may be parameterized. For example, $put(x, y)$ might stand for the action of putting object x on object y, in which case $do(put(A, B), s)$ denotes that situation resulting from placing A on B when the history is s. In the situation calculus, actions are denoted by first order terms, and situations (world histories) are also first order terms. For example, $do(putdown(A), do(walk(L), do(pickup(A), S_0)))$ is the situation denoting the world history consisting of the sequence of actions [pickup(A), walk(L), putdown(A)]. Notice that the sequence of actions in a history, in the order in which they occur, is obtained from a situation term by reading off the actions from right to left.

[1] We allow nondeterministic programs, so that, even by itself, this idea of an on-line execution of a program is problematic. See Section 3 below.

Relations whose truth values vary from situation to situation are called *relational fluents*. They are denoted by predicate symbols taking a situation term as their last argument. Similarly, functions whose values vary from situation to situation are called *functional fluents*, and are denoted by function symbols taking a situation term as their last argument. For example, $isCarrying(robot, p, s)$, meaning that a *robot* is carrying package p in situation s, is a relational fluent; $location(robot, s)$, denoting the location of *robot* in situation s, is a functional fluent. For simplicity, we shall not treat functional fluents in this paper.

To axiomatize the primitive actions and fluents of a domain of application, one must provide the following axioms:

1. Action precondition axioms, one for each primitive action $A(\vec{x})$, having the syntactic form
$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s),$$
where $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$, and whose only situation term is s. Action precondition axioms characterize (via the formula $\Pi_A(\vec{x}, s)$) the conditions under which it is possible to execute action $A(\vec{x})$ in situation s. In addition to these, one must provide suitable unique names axioms for actions.
2. Successor state axioms, one for each fluent F, having the 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 $\vec{x}, a, s$, and whose only situation term is s. Successor state axioms embody the solution to the frame problem of Reiter [21].
3. Axioms describing the initial situation – what is true initially, before any actions have occurred. This is any finite set of sentences that mention only the situation term S_0, or that are situation independent.

2.1 Golog

The "traditional" approach to high-level robotic control is to provide suitable goals, derive plans for achieving these goals, then have the robot execute these plans. Planning, however, is known to be computationally intractable in general, and in any case, is out of the question for deriving complex behaviors involving many hundreds, and possibly thousands of actions. The perspective being pursued by the Cognitive Robotics Group at the University of Toronto is to reduce the reliance on *planning* for eliciting interesting robot behaviors, and instead provide the robot with *programs* written in a suitable high-level language [16], in our case, *Golog* or *ConGolog*. As presented in [17] and extended in [8], *Golog* is a logic-programming language whose primitive actions are those of a background domain theory. Typically *Golog* programs are intended to be executed *off-line*, and then the sequence of actions returned by this off-line computation is executed on-line. Here we consider a variant of *Golog* that is intended to be executed entirely *on-line* [9]. It includes the following constructs:

nil,	empty program
a,	primitive action
$\phi?$,	test the truth of condition ϕ
$(\delta_1 ; \delta_2)$,	sequence
$(\delta_1 \mid \delta_2)$,	nondeterministic choice of two actions
$\pi v.\delta$,	nondeterministic choice of argument to an action
δ^*,	nondeterministic iteration
proc $P(\vec{v})\,\beta$ **end**,	procedure with formal parameters $\vec{v}$ and body β.

In contrast to straight line or partially ordered plans a *Golog* program can be arbitrary complex, including loops, recursive procedures and nondeterministic choice.

Example 2.1 The following is a blocks world Golog program that nondeterministically builds a tower of blocks spelling "paris" or "rome". In turn, the procedure for building a Rome tower nondeterministically determines a block with the letter "e" that is clear and on the table, then nondeterministically selects a block with letter "m" and moves it onto the "e" block, etc. There is a similar procedure for *makeParis*; neither procedure has any parameters.

proc $tower \quad makeParis \mid makeRome$ **endProc**.
proc $makeRome$
$\quad \pi\, b_0.[e(b_0) \wedge ontable(b_0) \wedge clear(b_0)]?\,;$
$\quad\quad \pi\, b_1.m(b_1)?\,; move(b_1, b_0)\,;$
$\quad\quad\quad \pi\, b_2.o(b_2)?\,; move(b_2, b_1)\,;$
$\quad\quad\quad\quad \pi\, b_3.r(b_3)?\,; move(b_3, b_2)$
endProc

proc $makeParis$
$\quad \pi\, b_0.[s(b_0) \wedge ontable(b_0) \wedge clear(b_0)]?\,;$
$\quad\quad \pi\, b_1.i(b_1)?\,; move(b_1, b_0)\,;$
$\quad\quad\quad \pi\, b_2.r(b_2)?\,; move(b_2, b_1)\,;$
$\quad\quad\quad\quad \pi\, b_3.a(b_3)?\,; move(b_3, b_2)$
$\quad\quad\quad\quad\quad \pi\, b_4.p(b_4)?\,; move(b_4, b_3)$
endProc

As in [8], we associate to programs a *transition semantics*, i.e. a semantics based on single steps of program execution. Informally, this semantics declares that as a program proceeds, a program counter moves from the very beginning of the program along its intermediate states. A *configuration* is a pair consisting of a program state (the part of the original program that is left to perform) and a situation.

To specify this semantics, we introduce two predicates *Trans* and *Final*.

- $Trans(\delta, s, \delta', s')$, given a program δ and a situation 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.
- $Final(\delta, s)$ tells us whether a configuration (δ, s) can be considered *final*, that is whether the computation is completed (no program remains to be executed). We have $Final(nil, s)$, but also $Final(\delta^*, s)$ since δ^* requires 0 or more repetitions of δ and so it is possible not to execute δ at all, completing the program immediately.

Trans

The predicate *Trans* is characterized by the following axioms:

1. Empty program:
$$Trans(nil, s, \delta', s') \equiv False$$
2. Primitive actions:
$$Trans(a, s, \delta', s') \equiv Poss(a, s) \wedge \delta' = nil \wedge s' = do(a, s)$$
3. Test actions:[2]
$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$$
4. Sequence:
$$Trans(\delta_1 ; \delta_2, s, \delta', s') \equiv \exists \gamma. Trans(\delta_1, s, \gamma, s') \wedge \delta' = \gamma ; \delta_2 \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')$$
5. Nondeterministic choice:
$$Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$$
6. Pick:
$$Trans(\pi v.\delta, s, \delta', s') \equiv \exists x. Trans(\delta^v_x, s, \delta', s')\ ^3$$
7. Iteration:
$$Trans(\delta^*, s, \delta', s') \equiv \exists \gamma. Trans(\delta, s, \gamma, s') \wedge \delta' = \gamma ; \delta^*$$

[2] We write ϕ to denote a term representing a situation calculus formula with suppressed situational argument and $\phi[s]$ to denote the formula with the restored argument. We assume any standard way of encoding first-order situation calculus formulas.

[3] Here, δ^v_x is the program resulting from substituting x for v uniformly in δ.

The assertions above characterize when a configuration (δ, s) can evolve (in a single step) to a configuration (δ', s'). Intuitively they can be read as follows:

1. (nil, s) cannot evolve to any configuration.
2. (a, s) evolves to $(nil, do(a, s))$, provided it is possible to execute a in s. Notice that after having performed a, nothing remains to be performed.
3. $(\phi?, s)$ evolves to (nil, s), provided that $\phi[s]$ holds. Otherwise, it cannot proceed. Notice that in any case the situation remains unchanged.
4. $(\delta_1; \delta_2, s)$ can evolve to $(\delta_1'; \delta_2, s')$, provided that (δ_1, s) can evolve to (δ_1', s'). Otherwise, it can evolve to (δ_2', s'), provided that (δ_1, s) is a final configuration and (δ_2, s) can evolve to (δ_2', s').
5. $(\delta_1|\delta_2, s)$ can evolve to (δ', s'), provided that either (δ_1, s) or (δ_2, s) can do so.
6. $(\pi v.\delta, s)$ can evolve to (δ', s'), provided that there exists an x such that (δ_x^v, s) can evolve to (δ', s').
7. (δ^*, s) can evolve to $(\delta'; \delta^*, s')$ provided that (δ, s) can evolve to (δ', s'). Observe that (δ^*, s) can also not evolve at all, because (δ^*, s) is final by definition (see below).

To simplify the discussion, we have omitted axioms for procedures. These can be found in the extended version of [8].

Final

The predicate *Final* is characterized by the following axioms:

1. Empty program:
$$Final(nil, s) \equiv True$$
2. Primitive action:
$$Final(a, s) \equiv False$$
3. Test action:
$$Final(\phi?, s) \equiv False$$
4. Sequence:
$$Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$
5. Nondeterministic choice:
$$Final(\delta_1 \mid \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$
6. Pick:
$$Final(\pi v.\delta, s) \equiv \exists x.Final(\delta_x^v, s)$$
7. Iteration:
$$Final(\delta^*, s) \equiv True$$

*Trans** **and** *Do*

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 of the following two sentences:

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

Using $Trans^*$ and *Final* we can give a new definition of the *Do* relation of [17] as:

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

In other words, $Do(\delta, s, s')$ holds iff it is possible to repeatedly single-step the program δ, obtaining a program δ' and a situation s' such that δ' can legally terminate in s'.

3 On vs. Off-Line Golog Interpreters

Before describing our approach to execution monitoring, we must first distinguish carefully between on-line and off-line *Golog* interpreters.[4] The relation $Do(\gamma, s, s')$ means that s' is a terminating situation resulting from an execution of program γ beginning with situation s. This relation has a natural Prolog implementation in terms of the one-step interpreter *trans*:

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

A Brave On-Line Interpreter

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

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

[4] An on-line interpreter based on *Trans* and *Final* was originally proposed in [9] to give an account of Golog /ConGolog programs with sensing actions. Here we make use of a simplified on-line interpreter that does not deal with sensing actions, but is suitable for coupling with an execution monitor.

The on and off-line interpreters differ only in the latter's use of the Prolog cut (!) to prevent backtracking to `trans` to select an alternative first action of `Prog`.[5] The effect is to commit to the first action selected by `trans`. We need this because a robot cannot undo any actions that it has actually performed in the physical world. It is this commitment that qualifies the clause to be understood as on-line interpreter. We refer to it as *brave* because it may well reach a dead-end, even if the program it is interpreting has a terminating situation.

A Cautious On-Line Interpreter

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

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

This is much more cautious than its brave counterpart; it commits to a first action only if that action is guaranteed to lead to a successful off-line termination of the program. Provided this program has a terminating situation, a cautious on-line interpreter never reaches a dead-end.

A cautious on-line interpreter appeals to the off-line execution of the robot's program (in the process of guaranteeing that after committing to a program action, the remainder of the program terminates). Therefore, this requirement precludes cautious interpretation of robot programs that appeal to sensing actions [15], since such actions cannot be performed off-line.[6] Because the brave interpreter never looks ahead, it is suitable for programs with sense actions. The price it pays for this is a greater risk of following dead-end paths.

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

The above interpreters are implemented in Prolog, and are lifted directly from *Final*, *Trans*, and *Do* introduced above. Such interpreters require that the domain specific action precondition and successor state axioms, and axioms about the initial situation, be expressible as Prolog clauses. Therefore, our implementation inherits Prolog's Closed World Assumption, but this is a limitation of the implementation, not the general theory. The full version of the cautious on-line interpreter can be found in [10].

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

[6] However, one could imagine a cautious interpreter that verifies off-line that the program terminates for all possible outcomes of its sensing actions. Even better, perhaps the programmer has already proved this.

4 Execution Monitoring of Golog Programs

In this section we give a situation calculus specification for the behavior of a Golog program under the influence of an execution monitor. We first provide a very general framework, without committing to any particular details of the monitor. Then we describe one specific monitor that forms the basis for the implementation of Section 5 below.

4.1 The General Framework

Here we discuss how on-line interpretation of Golog programs can be combined with a monitor. We imagine that after executing a primitive action or evaluating a test condition, a robot compares its mental world model with reality. We assume that all discrepancies between the robot's mental world and reality are the result of exogenous actions, and moreover, that the robot observes all such actions.[7] It will be the execution monitor that observes whether an exogenous action has changed the values of one or several fluents and, if necessary, recovers from this unanticipated event. This cycle of on-line interpreting, sensing and recovering (if necessary) repeats until the program terminates.

Just as we specified a semantics, via $Trans$, for *Golog* programs in Section 2.1, we want now to specify such

[7] A similar idealization about the observability of all exogenous events is a common assumption in discrete event control theory (e.g. [20, 6]). On the face of it, this idealization seems dubious in practice. One can argue convincingly that agents never observe action *occurrences* – Fido ate the sandwich – only their *effects* – The sandwich is no longer on the table. One can reconcile both points of view by supposing that instead of directly sensing exogenous actions, the robot can sense only the truth values of certain fluents. One can then introduce a set of new fictitious actions, one for each such fluent, whose effects are to alter their corresponding fluents' truth values. The robot can compute, from its successor state axioms, what fluents hold in its mental world. Now, when the robot observes the values of some its fluents in the physical world, it compares them with their values in its mental world; all discrepancies, if any, can be determined directly. Then, it can determine which fictitious actions must have "occurred" to account for the observed discrepancies between the physical world and the robot's mental world. It is these (inferred) fictitious actions that assume the role of the observable exogenous actions mentioned above.

a semantics for *Golog* programs *with* execution monitoring. Our definition will parallel that of Section 2.1. This closed-loop system (online interpreter and execution monitor) is characterized formally by a new predicate symbol $TransEM(\delta, s, \delta', s')$, describing a one-step transition consisting of a single $Trans$ 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 an exogenous action and (if necessary) to generate a program to counter-balance any perceived discrepancy. As a result of all this, the system passes from configuration (δ, s) to configuration (δ', s') specified as follows:

$$TransEM(\delta, s, \delta', s') \equiv \exists \delta'', s''.Trans(\delta, s, \delta'', s'') \wedge Monitor(\delta'', s'', \delta', s').$$

This is a brave version of $TransEM$. Its cautious counterpart is:

$$TransEM(\delta, s, \delta', s') \equiv \exists \delta'', s''.Trans(\delta, s, \delta'', s'') \wedge \exists s'''.Do(\delta'', s'', s''') \wedge Monitor(\delta'', s'', \delta', s').$$

The possible configurations that can be reached by a program δ from a situation s with execution monitoring are those obtained by repeatedly following $TransEM$ transitions, i.e. those in the reflexive transitive closure of this relation.

As we did for implementing on-line *Golog* interpreters (Section 3), we can now describe brave and cautious versions of on-line *Golog* interpreters *with* execution monitoring.

Brave On-Line Execution Monitor

```
onlineEM(Prog,S0,Sf) :- final(Prog,S0), S0 = Sf ;
  trans(Prog,S0,Prog1,S1),
  !,
  monitor(Prog1,S1,Prog2,S2), !,
  onlineEM(Prog2,S2,Sf).
```

Cautious On-Line Execution Monitor

```
onlineEM(Prog,S0,Sf) :- final(Prog,S0), S0 = Sf ;
  trans(Prog,S0,Prog1,S1),
  offline(Prog1,S1,S),
  !,
  monitor(Prog1,S1,Prog2,S2), !,
  onlineEM(Prog2,S2,Sf).
```

Next, we focus on the monitor. Let exo be an exogenous event, which might be as simple as a primitive action, or as complex as an arbitrary *Golog* program. We specify the behavior of our generic monitor by:

$$\begin{aligned} Monitor(\delta, s, \delta', s') \equiv \exists exo.Do(exo, s, s') \wedge \\ [\neg Relevant(\delta, s, s') \wedge \delta' = \delta \vee \\ Relevant(\delta, s, s') \wedge Recover(\delta, s, s', \delta')]. \end{aligned} \tag{1}$$

Here, $Relevant(\delta, s, s')$ is a predicate that specifies whether the discrepancy between s and s' is relevant in the current state δ of the program. If this discrepancy does not matter – $\neg Relevant(\delta, s, s')$ – then the execution monitor takes no action – $\delta' = \delta$. Otherwise, the monitor should recover from the exogenous action. The predicate $Recover(\delta, s, s', \delta')$ provides for this by determining a new program, δ', whose execution in situation s' 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.

A wide range of monitors can be achieved by defining $Relevant$ and $Recover$ in different ways. In the next section we elaborate on one such choice, one that will form the basis of the implementation of Section 5.

4.2 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 that may or may not take into account exogenous actions occurring when the robot executes the program. We shall assume that this program has a particular form, one that takes into account the programmer's *goal* in writing it. Specifically, we assume that along with her program, the programmer provides a first order sentence describing the program's goal, or what programmers call a program *postcondition*. We assume further that this postcondition is postfixed to the program. In other words, if δ is the original program, and *goal* is its postcondition, then the program we shall be dealing with in this paper will be $\delta\,; goal?$. This may seem a useless thing to do whenever δ is known to satisfy its postconditions, but as we shall see below, our approach to execution monitoring will change δ, and we shall need a guarantee that whenever the modified program terminates, it does so in a situation satisfying the original postcondition.

Next, we take $Relevant(\delta, s, s')$ to be $\neg \exists s'' Do(\delta, s', s'')$, so that the definition (1) of $Monitor$ becomes:

$$\begin{aligned} Monitor(\delta, s, \delta', s') \equiv \exists exo.Do(exo, s, s') \wedge \\ [\exists s'' Do(\delta, s', s'') \wedge \delta' = \delta \vee \\ \neg \exists s'' Do(\delta, s', s'') \wedge Recover(\delta, s, s', \delta')]. \end{aligned}$$

$Monitor$ checks for the existence of an exogenous program, determines the situation s' reached by this program, and if the monitored program δ terminates offline, the monitor returns δ, else it invokes a recovery mechanism to determine a new program δ'. 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 in 2 above.)

It only remains to specify the predicate $Recover(\delta, s, s', \delta')$ that is true whenever δ is the current state of the program being monitored, s is the situation prior to the occurrence of the exogenous program, s' is the situation after the exogenous event, and δ' is a new program to be executed on-line in place of δ, beginning in situation s'. We adopt the following specifications of *Recover*; it forms the basis of the implementation to be described later:

$$
\begin{aligned}
&Recover(\delta, s, s', \delta') \equiv \\
&\quad \exists p.straightLineProg(p) \wedge \\
&\quad\quad \exists s''.Do(p\,;\delta, s', s'') \wedge \delta' = p\,;\delta \wedge \\
&\quad\quad [\forall p', s''.straightLineProg(p') \wedge Do(p'\,;\delta, s', s'') \supset \\
&\quad\quad\quad length(p) \leq length(p')].
\end{aligned}
$$

Here, the recovery mechanism is conceptually quite simple; it determines a shortest straight-line program p such that, when prefixed onto the program δ, yields a program that terminates off-line. This is quite easy to implement; in its simplest form, simply generate all length one prefixes, test whether they yield a terminating off-line computation, then all length two prefixes, etc, until one succeeds, or some complexity bound is exceeded.[8] Notice that here we are appealing to the assumption 2 above that all monitored programs are postfixed with their goal conditions. We need something like this because the recovery mechanism *changes* the program being monitored, by adding a prefix to it. The resulting program may well terminate, but in doing so, it may behave in ways unintended by the programmer. But so long as the goal condition has been postfixed to the original program, all terminating executions of the altered program will still satisfy the programmer's intentions.

One disadvantage of the above recovery mechanism is that it will not recognize instances of exogenous events that happen to *help* in achieving the goal condition. In the extreme case of this, an exogenous event might create a situation that actually satisfies the goal. The above recovery procedure, being blind to such possibilities, will unthinkingly modify the current program state by prefixing to it a suitable plan, and execute the result, despite the fact that in reality, it is already where it wants to be. In effect, the recovery procedure has a built-in assumption that all exogenous events, if not neutral with respect to achieving the goal, are malicious.

[8] One can imagine much more sophisticated realizations of this simple idea that make use of the actions performed by *exo*, but we do not pursue this topic here.

5 An Implementation

The above theory of execution monitoring is supported by an implementation, in Prolog, that we demonstrate here for the blocks world program of Example 2.1. We use the cautious on-line monitor of Section 4.1, and a straightforward implementation of *Monitor* and $straightLineProg(p)$. The Prolog code is provided in [10].

5.1 A Blocks World Example

In this section, the blocks world is axiomatized with successor state and action precondition axioms. We use the following function and predicate constants.

Actions

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

Fluents

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

Other predicate constants

The predicates $R(b)$, $O(b)$, $M(b)$, $E(b)$, $P(b)$, $A(b)$, $I(b)$, $S(b)$ are true when their arguments are blocks with the corresponding letters on them.

Successor state axioms

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

$$
\begin{aligned}
&Ontable(x, do(a, s)) \equiv a = moveToTable(x) \vee \\
&\quad Ontable(x, s) \wedge \neg(\exists y) a = move(x, y).
\end{aligned}
$$

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

Action precondition axioms

$$
\begin{aligned}
&Poss(move(x, y), s) \equiv Clear(x, s) \wedge \\
&\quad Clear(y, s) \wedge x \neq y.
\end{aligned}
$$

$$
\begin{aligned}
&Poss(moveToTable(x), s) \equiv Clear(x, s) \wedge \\
&\quad \neg Ontable(x, s).
\end{aligned}
$$

Unique names axioms for actions

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

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

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

In our example, the initial situation is such that all blocks are on the table and clear (see Figure 1). There is no block with the letter "p"[9], but there are several blocks with letters for spelling "aris" and "rome", as well as blocks with letters "n" and "f" (which are irrelevant to building a tower spelling "rome" or "paris").

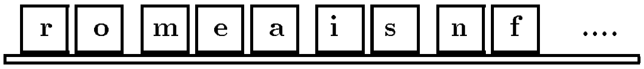

Figure 1: Part of the initial situation.

The program goal is:

$$Goal(s) \equiv SpellsParis(s) \vee SpellsRome(s),$$

$$\begin{aligned} SpellsRome(s) \equiv\ & (\exists b_0, b_1, b_2, b_3). \\ & R(b_3) \wedge O(b_2) \wedge M(b_1) \wedge E(b_0)\ \wedge \\ & \quad Ontable(b_0, s) \wedge On(b_1, b_0, s) \wedge \\ & \quad\quad On(b_2, b_1, s) \wedge On(b_3, b_2, s) \wedge Clear(b_3, s). \end{aligned}$$

$$\begin{aligned} SpellsParis(s) \equiv\ & (\exists b_0, b_1, b_2, b_3, b_4). \\ & P(b_4) \wedge A(b_3) \wedge R(b_2) \wedge I(b_1) \wedge S(b_0)\ \wedge \\ & \quad Ontable(b_0, s) \wedge On(b_1, b_0, s) \wedge On(b_2, b_1, s) \wedge \\ & \quad\quad On(b_3, b_2, s) \wedge On(b_4, b_3, s) \wedge clear(b_4, s). \end{aligned}$$

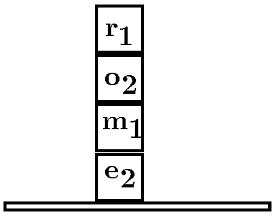

Figure 2: A goal arrangement of blocks.

Figure 2 represents an arrangement of blocks that satisfies the program goal.

An implementation, in Eclipse Prolog, is provided in [10].

5.2 An Execution Trace

The original procedure *tower* is very simple and was not designed to respond to external disturbances of any kind. However, as the trace demonstrates, the execution monitor is able to produce fairly sophisticated behavior in response to unforeseen exogenous events.

In Golog, tests do not change the situation, but all other primitive actions do. Each time the program performs a primitive action or evaluates a test, an exogenous program may occur. In the example below, any sequence of actions that is possible in the current situation can be performed by an external agent. This is realized in the implementation by interactively asking the user to provide exogenous events after each elementary program operation (test or primitive action).

The following is an annotated trace of the first several steps of our implementation for this blocks world setting. We use `this font` for the actual output of the program and *italics* for states of the *Golog* program *tower* of Example 2.1. The symbol ":" in the Prolog implementation of the interpreter corresponds to sequential composition ";" in Golog and Prolog's term "pi" corresponds to π in Golog, etc.

```
[eclipse] onlineEM((tower : ?(goal)), s0, S).

 Program state = (nil: pi(b1,?(m(b1)) : move(b1,e1):
    pi(b2,?(o(b2)) : move(b2,b1) : pi(b3,?(r(b3)) :
        move(b3,b2))))) : ?(goal)
 Current situation: s0
```

The cautious interpreter first tried to execute *makeParis* off-line. This failed because there is no "p" block. It then proceeded with *makeRome*.[10] According to the implementation of a cautious on-line interpreter (see section 3), *Trans* does the first step of *makeRome*:

$\pi\, b_0.[e(b_0) \wedge ontable(b_0) \wedge clear(b_0)]?\,;$

by determining that the block e_1 will be the base of a goal tower. The remainder of the program (the "program state" in the output above) is the following:

$$\begin{aligned} & nil\ ; \\ & \pi\, b_1.m(b_1)?\,; move(b_1, e1)\,; \\ & \quad \pi\, b_2.o(b_2)?\,; move(b_2, b_1)\,; \\ & \quad\quad \pi\, b_3.r(b_3)?\,; move(b_3, b_2)\ ;\ (goal)? \end{aligned}$$

where *nil* results after the first step (see axioms of *Trans* for $\pi v.\delta$ and $\phi?$).

```
>Enter: an exogenous program or noOp if none occurs.
         move(n,m1) : move(f,n) : move(i2,o3).

 No recovery necessary. Proceeding with the next
 step of program.

  Program state = (nil : move(m2, e1) :
     pi(b2, ?(o(b2)) : move(b2, m2) :
         pi(b3, ?(r(b3)) : move(b3, b2)))) : ?(goal)
  Current situation: do(move(i2, o3), do(move(f, n),
                   do(move(n, m1), s0)))
```

The first exogenous program covered blocks m_1, n and o_3 (see Fig. 3), but the remaining program

$nil\ ;$
$move(m_2, e_1)\,;\qquad \surd$

[9] We selected this arrangement intentionally to illustrate the difference between cautious and brave interpreters.

[10] A brave interpreter would have eventually failed, without even trying *makeRome*.

$$
\begin{array}{l}
\quad \pi\, b_2.o(b_2)?\,; move(b_2, m_2)\,; \\
\quad\quad \pi\, b_3.r(b_3)?\,; move(b_3, b_2)\ ;\ ?(goal)
\end{array}
$$

can still be successfully completed because there remain enough uncovered blocks of the right kind to construct "rome", so it continues.

```
>Enter: an exogenous program or noOp if none occurs.
          move(i1,o1) : move(r2,o2).
Start recovering...

 New program = moveToTable(r2) :
     (nil : move(m2,e1) : pi(b2,?(o(b2)) :
         move(b2,m2) : pi(b3,?(r(b3)) :
             move(b3,b2)))) : ?(goal)

 Current situation: do(move(r2,o2),do(move(i1,o1),
                 do(move(i2,o3), do(move(f,n),
                    do(move(n,m1),s0)))))
```

After the second exogenous program `move(i1,o1) : move(r2,o2)`, all three blocks with letter "o" are covered (see Figure 4).

Because it is not possible to move blocks o_1, o_2, o_3 (by the precondition axiom for $move(x, y)$), the remaining program cannot be completed. Hence, the $Monitor$ gives control to $Recover$ that, with the help of the planner $straightLineProg(p)$, determines a shortest corrective sequence p of actions (namely `moveToTable(r2)`) in order to allow the program to resume, and prefixes this action to the previous program state:

$$
\begin{array}{l}
moveToTable(r2)\,; nil\ \ ; \\
\quad move(m_2, e_1)\,; \\
\quad\quad \pi\, b_2.o(b_2)?\,; move(b_2, m_2)\,; \\
\quad\quad\quad \pi\, b_3.r(b_3)?\,; move(b_3, b_2)\ ;\ ?(goal)
\end{array}
$$

From this point on, the on-line evaluation continues by doing one step of the new program. If after that, no exogenous disturbances occur during the next two steps of the execution of this new program, it will reach the following program state:

$$
\begin{array}{l}
nil\,; \\
\quad move(o2, m2)\,; \\
\quad\quad \pi\, b_3.r(b_3)?\,; move(b_3, o_2)\,; ?(goal)
\end{array}
\tag{2}
$$

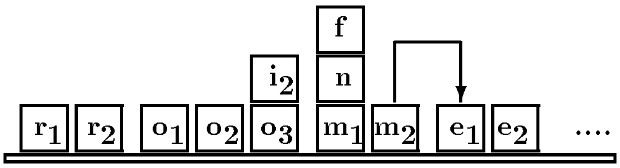

Figure 3: The first exogenous disturbance occurred when the program was ready to move m_2 on top of e_1.

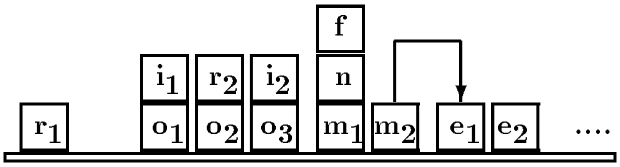

Figure 4: The second exogenous disturbance.

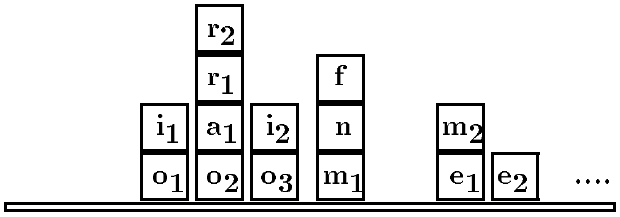

Figure 5: The pile of blocks covers the block o_2.

Let assume that at this point a third exogenous program occurs (see Figure 5):

```
>Enter: an exogenous program or noOp if none occurs.
         move(a1,o2) : move(r1,a1) : move(r2,r1).
Start recovering...
  New program = (moveToTable(r2) :
      moveToTable(r1) : moveToTable(a1)) :
         (nil : move(o2, m2) : pi(b3, ?(r(b3)) :
            move(b3, o2))) : ?(goal)

  Program state = (nil : moveToTable(r1) :
     moveToTable(a1)) : (nil : move(o2, m2) :
        pi(b3, ?(r(b3)) : move(b3, o2))) : ?(goal)

  Current situation: do(moveToTable(r2),
    do(move(r2, r1), do(move(r1, a1),
      do(move(a1, o2), do(move(m2, e1),
        do(moveToTable(r2), do(move(r2, o2),
         do(move(i1, o1), do(move(i2, o3),
           do(move(f,n), do(move(n,m1), s0))))))))))
```

The recovery procedure determined that the sequence of three corrective actions `moveToTable(r2)`, `moveToTable(r1)`, `moveToTable(a1)` will lead to a situation where the program (2) can be resumed, and moreover, this is a shortest such correction. If no other exogenous actions happen, the program will eventually successfully terminate, having built a tower for "rome".

6 Correctness Properties for Execution Monitors

With definitions for $TransEM$ and $Monitor$, as in Section 4.1, it becomes possible to formulate, and ultimately prove, various correctness properties for execution monitors. These properties are intended to capture suitable concepts of *controllability* following the intuition behind similar concepts introduced for discrete event systems [20]. Informally, controllability is the property that characterizes a closed-loop system (a *Golog* program coupled with the execution monitor): this is the ability of a monitored program to behave correctly even if exogenous actions occur during the robot's execution of the program. There are many possible definitions, with varying degrees of generality, of what counts as correct behavior of a monitored system. We focus here on various correctness properties one might want to prove of the monitor.

Recall that the general execution monitor, as specified by (1), is the relation $Monitor(\delta, s, s', \delta')$, meaning that whenever δ is the state of monitored program in situation s, and s' is a situation resulting from an exogenous event occurrence at s, then the on-line execution should resume in s' with the new program δ'. Then, by analogy with Floyd-Hoare style correctness and termination properties, we can formulate a variety of verification tasks, some examples of which we now describe. These are parameterized by two predicates:

1. $P(s)$, a desirable property that a terminating situation s must satisfy. For example, P might describe a postcondition of the program being monitored.
2. $Q(\delta, s, s')$, a relationship between the current program state δ and s, the current situation, and s', the situation resulting from an exogenous event occurring in s. For example, Q might express that δ terminates off-line when executed in s and also when executed in s'.

Weak Termination and Correctness

$$(\forall \delta, s, s').Q(\delta, s, s') \supset Monitor(\delta, s, s', \delta') \supset (\exists s'').Do(\delta', s', s'') \wedge P(s'').$$

The task here is to verify that, under condition Q, whenever *Monitor* determines a new program with which to resume the system computation after an exogenous event occurrence, that program has a terminating (off-line) computation resulting in a final situation in which P holds.

Strong Termination and Correctness

$$(\forall \delta, s, s').Q(\delta, s, s') \supset (\exists \delta').Monitor(\delta, s, s', \delta') \wedge (\exists s'').Do(\delta', s', s'') \wedge P(s''),$$

Under condition Q, *Monitor* always determines a new program with which to resume the system computation after an exogenous event occurrence, and that program has a terminating (off-line) computation resulting in a final situation in which P holds.

Even Stronger Termination and Correctness

$$(\forall \delta, s, s').Q(\delta, s, s') \supset (\exists \delta').Monitor(\delta, s, s', \delta') \wedge (\exists s'').Do(\delta', s', s'') \wedge (\forall s'').Do(\delta', s', s'') \supset P(s'').$$

Here, the correctness property is that under condition Q, *Monitor* always determines a new program that terminates off-line, and all these terminating situations satisfy P.

It is also possible to formulate various correctness properties for the entire monitored system, for example, the weak property that provided the monitored program terminates, then it does so in a desirable situation:

$$(\forall \delta, s, s').TransEM^*(\delta, s, nil, s') \supset P(s'),$$

where $TransEM^*$ is the transitive closure of $TransEM$.

Other variations on the above themes are possible, but our purpose here is not to pursue these issues in depth, but simply to point out that correctness properties for monitored systems are easily formulated within our framework. Moreover, because this framework is entirely within the situation calculus, such correctness proofs can be constructed totally within a classical logic.

7 Discussion

There are several systems designed to interleave monitoring with plan execution: PLANEX [7], IPEM [1], ROGUE [13], SPEEDY [3]. We differ from these and similar proposals, first by the formal neatness of our approach, 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 provided by a programmer.

In a sequence of papers [25, 26, 27] Schoppers proposes and defends the idea of "universal plans", which "address the tension between reasoned behavior and timely response by caching reactions for classes of possible situations". From our point of view, the notion of a universal plan is closely related to the notion of controllable languages developed for discrete event systems control [20]. There, a language (a set of linear plans) is controllable iff the effects of all possible uncontrollable events do not lead outside the set of plans that this language contains. In other words, just as for Schoppers, all required system reactions to possible contingencies are compiled into the controllable language. Our framework is different, but complementary; it favors the on-line generation of appropriate reactions to exogenous events, as opposed to precompiling them into the *Golog* program. *ConGolog* [8, 9] is a much richer version of *Golog* that supports concurrency, prioritized interrupts and exogenous actions. Reactive behaviors are easily representable by ConGolog's interrupt mechanism, so that a combination of reactive behaviors with "deliberative" execution monitoring is possible. This would allow one to experiment with different mixtures of execution monitoring and reactivity, with the advantage of preserving the unifying formal framework of the situation calculus, but this remains an open research problem.

The theory of embedded planning [29, 12, 30] introduces notions of planning with failure and has motivations very similar to ours. The authors propose several formal languages that, like *Golog*, include constructs for sequence, conditionals, loops and recursion. The emphasis is on reactive programs, but their proposal does provide for replanning during execution.

Several authors rely on formal theories of actions for the purposes of characterizing appropriate notions of action failures [2, 24], but they do not consider execution monitoring per se.

Perhaps the most sophisticated existing plan execution monitor is the XFRM system of Beetz and McDermott [4, 5]. This provides for the continual modification of robot plans (programs) during their execution, using a rich collection of failure models and plan repair strategies. Nothing in our proposal so far can rival the functionality of the XFRM system. The primary objective of our approach is to provide compact, declarative representations for the entire process of execution, monitoring and recovery from failure, and this paper has presented our framework for this research program. It remains to see whether the logical purity of our approach will pay off with clear, analyzable specifications that lead to implementations rivaling that of XFRM.

8 Conclusions and Future Work

We have presented a very general, situation calculus-based account of execution monitoring for high-level robot programs, and illustrated the theory with an implementation (in simulation mode) of a specific monitor. The approach has the advantage of being completely formal, and therefore is suitable for formulating, and ultimately proving, correctness properties for monitored systems.

Plans for ongoing and future work include the following issues:

1. Draw closer parallels with the concept of controllable systems in discrete event control theory [20, 6, 14].
2. Explore realizations of execution monitors different than that presented in Section 4.2 [28].
3. Investigate techniques for proving correctness properties of various monitors and monitored systems.
4. Investigate the concept of execution monitoring for non-terminating *Golog* programs [11].
5. Extend these ideas to temporal domains, for example, monitoring robot control programs written in sequential, temporal *Golog* [23].
6. Implement these ideas on the Cognitive Robotics Group's RWI B21 autonomous robot at the University of Toronto.

Acknowledgements:

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada, the Institute for Robotics and Intelligent Systems of the Government of Canada, and the Information Technology Research Centre of the Government of Ontario. Our thanks to the many people who contributed their ideas and suggestions during the evolution of our thinking about these issues: Yves Lespérance, Hector Levesque, Jeff Lloyd, John Mylopoulos, Brian Nixon, Fiora Pirri, Eric Sandewall, Murray Shanahan, Eugenia Ternovskaia, Murray Wonham.

References

[1] J. Ambrose-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *AAAI-88*, p. 735–740. Morgan Kaufmann Publishers, San Francisco, CA, 1988.

[2] C. Baral, T. and Son. Relating theories of actions and reactive control. In *Robots, Softbots, Immobots: Theories of Action, Planning and Control*, working notes of the workshop held on July 27, 1997 in conjunction with the AAAI-97, Providence, Rhode Island.

[3] C. Bastié and P. Régnier. SPEEDY : Monitoring the Execution in Dynamic Environments. In *Reasoning about Actions and Planning in Complex Environments*, Proceedings of the Workshop at International Conference on Formal and Applied Practical Reasoning, Bonn, Germany, on June 4, 1996. Available as: Technical Report AIDA-96-11, Fachgebiet Intellektik, Technische Hochschule Darmstadt, Germany.

[4] M. Beetz and D. McDermott. Improving robot plans during their execution. In *2nd Int. Conf. on AI Planning Systems (AIPS-94)*, Kris Hammond (editor), p. 3–12, 1994.

[5] M. Beetz and D. McDermott. Expressing transformations of structured reactive plans. In *4th European Conference on Planning (ECP'97)*, Sam Steel (editor), p. 66–78, Toulouse, France, September 24-26, 1997.

[6] S.-L. Chung, S.Lafortune, F. Ling. Limited lookahead policies in supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, volume 37, N 12, p. 1921–1935, 1992.

[7] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, volume 3, N 4, p. 251–288, 1972.

[8] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent executions, prioritized interrupts, and exogenous actions in the situation calculus. In *15th International Joint Conference on Artificial Intelligence (IJCAI–97)*, p. 1221–1226 1997.

[9] G. De Giacomo and H.J. Levesque. Congolog incremental interpreter. Technical report, Computer Science Department, University of Toronto, 1998. In preparation.

[10] G. De Giacomo, R. Reiter, M. Soutchanski Execution Monitoring of High-Level Robot Programs. In working notes of the 4th Symposium on *Logical Formalizations of Commonsense Reasoning*, London, UK, Jan. 6 - Jan. 9, 1998. Electronic version is available at: `www.ida.liu.se/ext/etai/nj/fcs-98/listing.html`

[11] G. De Giacomo, R. Reiter, E. Ternovskaia. Nonterminating processes in the situation calculus. In *Robots, Softbots, Immobots: Theories of Action, Planning and Control*, working notes of the workshop held on July 27, 1997 in conjunction with the AAAI-97, Providence, Rhode Island.

[12] F. Giunchiglia, P. Traverso, L. Spalazzi. Planning with failure. In 2nd International Conference on AI Planning Systems (AIPS-94), Chicago, IL, June 15-17, 1994.

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

[14] J. Kosecka and R. Bajcsy. Discrete Event Systems for Autonomous Mobile Agents. *Robotics and Autonomous Systems*, volume 12, p. 187–198, 1994.

[15] H.J. Levesque. What is planning in the presence of sensing? In *AAAI-96*, volume 2, p. 1139–1145, Portland, Oregon, 1996.

[16] H.J. Levesque and R. Reiter. High-level robotic control: beyond planning. In: *Integrating Robotics Research: Taking the Next Big Leap.* Position paper. AAAI 1998 Spring Symposium, Stanford University, March 23-25, 1998. Available at `http://www.cs.toronto.edu/~cogrobo/`

[17] 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*, 1997, volume 31, N 1–3, p. 59–83.

[18] F. Lin and R. Reiter. State constraints revisited. In *J. of Logic and Computation, special issue on actions and processes*, 1994, volume 4, p. 655–678.

[19] J.McCarthy, P.Hayes Some philosophical problems from the standpoint of artificial intelligence. In: B.Meltzer and D.Michie (editors), *Machine Intelligence*, volume 4, p. 463–502, Edinburgh University Press, 1969.

[20] P.J. Ramadge and W.M. Wonham. The Control of Discrete Event Systems. Proceedings of the IEEE, 77(1): 81–98, 1989.

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

[22] R. Reiter. KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems. A draft of the first eight chapters of a book. Available at `http://www.cs.toronto.edu/~cogrobo/`

[23] R. Reiter. Sequential, temporal Golog. In *Principles of Knowledge Representation and Reasoning.* Proceedings of the 6th International Conference (KR'98), Trento, Italy, June 2-5, 1998, this volume.

[24] E. Sandewall. Logic-Based Modelling of Goal-Directed Behavior. In *Linköping Electronic Articles in Computer and Information Science*, volume 2, N 19, 1997. Available at: `http://www.ep.liu.se/ea/cis/1997/019/`

[25] M.J. Schoppers. Universal plans for reactive robots in unpredictable environments In *10th International Joint Conference on Artificial Intelligence (IJCAI–87)*, p. 1039–1046, 1987.

[26] M.J. Schoppers. In defense of reaction plans as caches. In *AI Magazine*, volume 10, N 4, p. 51–60, 1989.

[27] M.J. Schoppers. Building monitors to exploit open-loop and closed-loop dynamics. In Proceedings of the 1st *International Conference on Artificial Iintelligence Planning Systems (AIPS-92)*, June 15-17, College Park, Maryland, p. 204–213, 1992

[28] M. Soutchanski. Execution monitoring of high-level robot programs. University of Toronto, PhD thesis, forthcoming.

[29] P. Traverso, A. Cimatti, L. Spalazzi. Beyond the single planning paradigm: introspective planning. In *10th European Conference on Artificial Intelligence (ECAI-92)*, p. 643–647, Vienna, August 3-7, 1992.

[30] P. Traverso and L. Spalazzi. A Logic for Acting, Sensing and Planning. In *14th International Joint Conference on Artificial Intelligence*, volume 2, p. 1941–1949, 1995.