Execution Monitoring of High-Level Robot Programs.

Giuseppe De Giacomo
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
degiacomo@dis.uniroma1.it

Ray Reiter
Department of Computer Science
University of Toronto
reiter@cs.toronto.edu

Mikhail Soutchanski*
Department of Computer Science
University of Toronto
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.

*Names of the authors are mentioned alphabetically.

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. Other recovery strategies are also possible, and will be discussed below.

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 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

[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.

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 situation calculus ([8]) is specifically designed for representing dynamically changing worlds. The version of the situation calculus that we use here has been described in [12], [7], and elsewhere. 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 $\Phi_F(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$, and whose only situation term is s. Successor state axioms embody the solution to the frame problem of Reiter [11].

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

As presented in [6] and extended in [3], *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 a sequence of actions should be extracted from such off-line computation and executed on-line. Here we consider a variant of *Golog* that is intended to be executed entirely *on-line* [4]. 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 the body β.

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 nondeterministcally determines a block with the letter "e" that is clear and on the table, then nondeterministcally selects a block with letter "m" and moves it onto the "e" block, etc. There is a similar procedure for *makeParis*. Note that all procedures do not have 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 [3], 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.

We introduce two predicates *Trans* and *Final*.[2]

[2]Axioms for procedures will be given in the extended version of [3].

- *Trans*(δ, s, δ', 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*(δ, s, δ', s') denotes a transition relation between configurations.

- *Final*(δ, 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). Obviously, we have *Final*(nil, s), but also *Final*(δ^*, 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:
$$Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s)$$

3. Test actions:[3]
$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$$

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:
$$Trans(\pi v.\delta, s, \delta', s') \equiv \exists x.\, Trans(\delta^v_x, s, \delta', s')$$

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

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:

[3] 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 of first-order situation calculus formulas.

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; here $\phi[s]$ is the result of restoring the suppressed situation arguments of all fluents mentioned by ϕ. 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'). Here, δ_x^v is the program resulting from substituting x for v uniformly in δ.

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).

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:
$$Final(\delta^*, s) \equiv True$$

TransCl **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 *TransCl*, is defined as the (second-order) situation calculus formula:

$$TransCl(\delta, s, \delta', s') \quad \equiv \quad \forall T[\ldots \quad \Rightarrow \quad T(\delta, s, \delta', s')]$$

where ... stands for the conjunction of the universal closure of the following two sentences:

$$\begin{array}{l} T(\delta, s, \delta, s) \\ Trans(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \quad \Rightarrow \quad T(\delta, s, \delta', s') \end{array}$$

Using *TransCl* and *Final* we can give a new definition of the *Do* relation of [6] as:

$$Do(\delta, s, s') \quad \equiv \quad \exists \delta'. TransCl(\delta, s, \delta', s') \wedge Final(\delta', s').$$

In other words, $Do(\delta, s, s')$ holds if 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(P,S,Sf) :- transCl(P,S,Pf,Sf),final(Pf,Sf).
transCl(P,S,P,S).
transCl(P,S,P2,S2) :- trans(P,S,P1,S1), transCl(P1,S1,P2,S2).
```

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 [4] 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`. 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 eventually 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 [14], 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.

The cautious on-line interpreter is implemented in Prolog. The interpreter is lifted directly from *Final*, *Trans*, and *Do* introduced above. Such an interpreter requires that the program's precondition axioms, successor state axioms, and axioms about the initial state be expressible as Prolog clauses. Therefore, our implementation inherits Prolog's Closed World Assumption. This is a limitation of the implementation, not the theory. The full version of the cautious on-line interpreter is included in the Appendix 1.

4 Execution Monitoring of Golog Programs

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 program test action, 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

[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.

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.

We assume 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.

Just as we specified a semantics, via $Trans$, for off-line Golog programs in Section 2.1, we want now to specify such 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 single step, 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:

$$\begin{aligned} TransEM(\delta, s, \delta', s') \equiv \exists \delta'', s''.&Trans(\delta, s, \delta'', s'') \wedge \\ &Monitor(\delta'', s'', \delta', s'). \end{aligned} \tag{1}$$

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.

[7] 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 posing the observation of action occurrences as an abduction problem: Given an observation of one or more effects, hypothesize an exogenous action occurrence to account for the observation. This is the implicit stance we take in this paper when we suppose that the robot has "observed" an exogenous action occurrence. By doing so, we are glossing over all the obvious technical problems associated with realizing this abductive point of view.

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), /* Select a first action of Prog. */
  !,                       /* Commit to this action. */
  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), /* Select a first action of Prog. */
  offline(Prog1,S1,S),  /* Make sure the rest of Prog terminates. */
  !,                    /* Commit to this action. */
  monitor(Prog1,S1,Prog2,S2),
  onlineEM(Prog2,S2,Sf).
```

4.1 A Monitor

Here, we elaborate on the definition (1) by specifying a class of monitors:

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

Monitor checks for the existence of an exogenous program, determines the situation s' reached by this program, and if the monitored program δ terminates off-line, 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 take control, so long as this is guaranteed to terminate off-line in the situation where the program's goal holds. Notice here that we have allowed for exogenous *programs*, not simply primitive actions. Other classes of monitors are certainly possible, but we do not pursue this question here.

4.2 A Recovery Procedure

The monitor of the previous section appeals to a relation $Recover(\delta, exo, s, \delta')$ that is true whenever δ is a program being monitored, *exo* is an exogenous program, s is the current situation after the occurrence of *exo*, and δ' is a new program to be executed on-line in place of δ, beginning in situation s. There are many possible specifications of *Recover*; the following is one that forms the basis of the implementation to be described later:

$$Recover(\delta, exo, s, \delta') \equiv$$
$$\exists p.straightLineProgram(p) \wedge \exists s'.Do(p\,;\delta, s, s') \wedge \delta' = p\,;\delta \wedge$$
$$[\forall p', s'.straightLineProgram(p') \wedge Do(p'\,;\delta, s, s') \supset$$

$$length(p) \leq length(p')].$$

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 it is here, and only here, that we appeal to the assumption 2 of Section 4 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.

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, and a straightforward implementation of $Monitor$ and $straightLineProgram(p)$. In the implementation, $recover(M, N, \delta, exo, s, \delta')$ has two numerical arguments in addition to 4 arguments mentioned above: M equals the minimal length of a prefix, and N equals the maximal possible length of the prefix, in other words, N is the complexity bound that should not be exceeded. Despite that the current implementation does not use the value of exo, it is included because in the general case, the recovery procedure may use it. Appendix 2 contains the Prolog code of an implementation of the monitor.

5.1 A Blocks-World Example.

In this section, the blocks world is axiomatized with successor state and action precondition axioms.

Successor State Axioms.

$$on(X, Y, do(A, S)) \equiv A = move(X, Y) \vee on(X, Y, S) \wedge A \neq moveToTable(X) \wedge A \neq move(X, Z).$$
$$ontable(X, do(A, S)) \equiv A = moveToTable(X) \vee ontable(X, S) \wedge A \neq move(X, Y).$$
$$clear(X, do(A, S)) \equiv (A = move(Y, Z) \vee A = moveToTable(Y)) \wedge on(Y, X, S) \\ \vee clear(X, S) \wedge A \neq move(Y, X).$$

[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.

Action Precondition Axioms.

$$poss(move(X,Y),S) \equiv clear(X,S) \wedge clear(Y,S) \wedge X \neq Y.$$
$$poss(moveToTable(X),S) \equiv clear(X,S) \wedge \neg ontable(X,S).$$

In the initial situation, all blocks are on the table and clear. There is no block with the letter "p", but there are several blocks with letters for spelling "aris" and "rome", as well as blocks with letters "n" and "f" (which are irrelevant to building a tower spelling "rome" or "paris"). The program goal is (omitting the obvious $spellsParis$ relation):

$$goal(s) \equiv spellsParis(s) \vee spellsRome(s),$$

$$spellsRome(s) \equiv (\exists b_0, b_1, b_2, b_3).r(b_3) \wedge o(b_2) \wedge m(b_1) \wedge e(b_0) \wedge$$
$$ontable(b_0,s) \wedge on(b_1,b_0,s) \wedge on(b_2,b_1,s) \wedge on(b_3,b_2,s) \wedge clear(b_3,s).$$

Note that because we specify what the goal is, our execution monitor will never try to recover from an exogenous action that moves "n" on "f" and which is completely irrelevant to the program.

An implementation with suitable Prolog clauses is provided in the Appendix 3.

5.2 An Execution Trace.

The original procedure *tower* is very simple and was not designed to deal with any kind of external disturbances. However, as the trace demonstrates, the execution monitor is able to produce fairly sophisticated behavior when it deals with exogenous unforeseen programs.

Note that in Golog, tests do not change the situation, but all other primitive actions do result in a new situation. 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. In the current implementation, the user simulates the activity of that external agent.

The following is an annotated trace of the first two steps of our implementation, in Eclipse Prolog, for this blocks world setting:

```
[eclipse] build.
 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.  A brave interpreter would have
    eventually failed, without even trying makeRome. */

>> Enter an exogenous program, or noop if none occurs.
```

```
       move(n,m1) : move(f,n) : move(i2,o3).

 No recovery necessary. Proceeding with next step of program.

/*  The exogenous program covered blocks m1, n and o3, but there are
    still enough uncovered blocks of the right kind to allow the
    current program state to construct "rome", so it continues.  */

 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)))

>> Enter an exogenous program, or noop if none occurs.
       move(i1,o1) : move(r2,o2).

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

Start recovering...

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

/*  After the exogenous program, all three blocks with letter "o" are
    covered. The recovery procedure determined the minimal actions
    (namely moveToTable(r2)) in order to allow the program to resume,
    and prefixed this to the previous program state. From this point,
    the on-line evaluation continues by doing one step of the new
    program, etc.  */
```

6 Conclusions and Future Work

The more elaborated version of this paper will discuss the relationship of our work to "traditional" approaches in AI to execution monitoring (e.g. PLANEX [2], IPEM [1], ROGUE [5]). We remark here only that we differ from these, 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.

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 [9, 10].
2. Prove soundness and/or completeness of various recovery procedures.

3. Extend these ideas to temporal domains, for example, monitoring robot control programs written in sequential, temporal Golog [13].
4. Implement these ideas on the Cognitive Robotics Group's RWI B21 autonomous robot at the University of Toronto.

References

[1] J. Ambrose-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'88)*, pages 735–740. Morgan Kaufmann Publishers, San Francisco, CA, 1988.

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

[3] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent executions, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI–97)*, 1997.

[4] G. De Giacomo and H.J. Levesque. Congolog incremental interpreter. Technical report, Computer Science Department, University of Toronto, 1997. In preparation.

[5] K.Z. Haigh and M. Veloso. Imterleaving planning and robot execution for asynchronous user requests. In *Proc. Int. Conf. on Intelligent Robots and Systems (IROS)*, 1996.

[6] 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, 59–83.

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

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

[9] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[10] P.J. Ramadge and W.M. Wonham. Modular feedback logic for discrete event systems. *SIAM Journal of Control and Optimization*, 25(5):1202–1218, 1987.

[11] 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*, pages 359–380. Academic Press, San Diego, CA, 1991.

[12] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

[13] R. Reiter. Sequential, temporal Golog. Technical report, Department of Computer Science, University of Toronto, 1997. http://www.cs.toronto.edu/˜cogrobo/.

[14] R. Scherl and H.J. Levesque. The frame problem and knowledge producing actions. In *Proc. AAAI-93*, pages 689–695, Washington, DC, 1993.

Appendix 1: on-line Golog interpreter

```
/*************************************************************************/
/*                        On-line Golog Interpreter                     */
/*************************************************************************/

:- op(800, xfy, [&]).   /* Conjunction */
:- op(850, xfy, [v]).   /* Disjunction */
:- op(870, xfy, [=>]).  /* Implication */
:- op(880,xfy, [<=>]).  /* Equivalence */
:- op(950, xfy, [:]).   /* Action sequence.*/
:- op(960, xfy, [#]).   /* Nondeterministic action choice.*/

/* trans(Prog,Sit,Prog_r,Sit_r) */

trans(A,S,nil,do(A,S)) :- primitive_action(A), poss(A,S).

trans(?(C),S,nil,S) :- holds(C,S).

trans(P1 : P2,S,P2r,Sr) :- final(P1,S),trans(P2,S,P2r,Sr).
trans(P1 : P2,S, P1r : P2,Sr) :- trans(P1,S,P1r,Sr).

trans(P1 # P2,S,Pr,Sr) :- trans(P1,S,Pr,Sr) ; trans(P2,S,Pr,Sr).
```

```
trans(pi(V,P),S,Pr,Sr) :- sub(V,_,P,PP), trans(PP,S,Pr,Sr).

trans(star(P),S,PP : star(P),Sr) :- trans(P,S,PP,Sr).

trans(if(C,P1,P2),S,Pr,Sr) :-  trans(((?(C) : P1) # (?(-C) : P2)),S,Pr,Sr).

trans(while(C,P),S,Pr,Sr) :-  trans(star(?(C) : P) : ?(-C), S,Pr,Sr).

trans(P,S,Pr,Sr) :- proc(P,Pbody), trans(Pbody,S,Pr,Sr).


/* final(Program,Situation) */

final(nil,S).

final(P1 : P2,S) :- final(P1,S),final(P2,S).

final(P1 # P2,S) :- final(P1,S) ; final(P2,S).

final(pi(V,P),S) :- final(P,S).

final(star(P),S).

final(if(C,P1,P2),S) :-
      final(( (?(C) : P1) # (?(-C) : P2) ),  S).

final(while(C,P),S) :-
      final( star(?(C) : P) : ?(-C) ,S).

final(pcall(P_Args),S) :- proc(P_Args,P),final(P,S).

/* transCl(Prog,S,Prog_r,S_r) is the transitive closure of trans(Prog,S,Prog_r,S_r) */

transCl(P,S,P,S).
transCl(P,S,P2,S2) :- trans(P,S,P1,S1), transCl(P1,S1,P2,S2).

/* offline(Prog,S,Sr): Sr is the situation resulting after doing Prog */

offline(P,S,Sr) :- transCl(P,S,Pr,Sr),final(Pr,Sr).

online(Delta,S,S2) :-  final(Delta,S), S2=S;
                   trans(Delta,S,Delta1,S1),
                  offline(Delta1,S1,Sg), /*  Delta1 leads to the goal */
                    !,
```

```
                  monitor(Delta1,S1,Delta2,S2),
                  online(Delta2,S2,S3).

/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New. */

sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                    T2 =..[F|L2].
sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

/* The holds predicate implements the revised Lloyd-Topor
   transformations on test conditions.  */

holds(P & Q,S) :- holds(P,S), holds(Q,S).
holds(P v Q,S) :- holds(P,S); holds(Q,S).
holds(P => Q,S) :- holds(-P v Q,S).
holds(P <=> Q,S) :- holds((P => Q) & (Q => P),S).
holds(-(-P),S) :- holds(P,S).
holds(-(P & Q),S) :- holds(-P v -Q,S).
holds(-(P v Q),S) :- holds(-P & -Q,S).
holds(-(P => Q),S) :- holds(-(-P v Q),S).
holds(-(P <=> Q),S) :- holds(-((P => Q) & (Q => P)),S).
holds(-all(V,P),S) :- holds(some(V,-P),S).
holds(-some(V,P),S) :- not holds(some(V,P),S).  /* Negation */
holds(-P,S) :- isAtom(P), not holds(P,S).     /* by failure */
holds(all(V,P),S) :- holds(-some(V,-P),S).
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).

/* The following clause treats the holds predicate for non fluents, including
   Prolog system predicates. For this to work properly, the Golog programmer
   must provide, for all fluents, a clause giving the result of restoring
   situation arguments to situation-suppressed terms, for example:
         restoreSitArg(ontable(X),S,ontable(X,S)).             */

holds(A,S) :- restoreSitArg(A,S,F), F ;
              not restoreSitArg(A,S,F), isAtom(A), A.

isAtom(A) :- not (A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
    A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W)).

restoreSitArg(poss(A),S,poss(A,S)).
```

Appendix 2: the monitor and recovery procedure

```
/*********************************************************************/
/*          The execution monitor calls recovery procedure         */
/*            to counter-balance external disturbances.             */
/*********************************************************************/

/* We allow EXOs be arbitrary Golog programs; hence, if Exo happens
   in S, then S1 is the situation resulting after termination of Exo */

monitor(DeltaOld,S,DeltaNew,S1) :- printMessages(DeltaOld,S),
       sense(Exo,S),
        ( Exo=noop -> (S1=S, DeltaNew=DeltaOld); /*No exogenous disturbances */
             /* Otherwise, simulate execution of exogenous */
           offline(Exo,S,S1),
             !,  /* Exogenous program cannot be redone */
          /* Determine whether the reminder of the program will
            terminate in the goal situation  */
           ( offline(DeltaOld,S1,Sg),
             nl,
        write('No recovery necessary. Proceeding with the next step of program. '),
             nl,
             DeltaNew=DeltaOld;
                /* otherwise, recover */
             write('Start recovering...'),
             recover(1,4,DeltaOld,Exo,S1,DeltaNew) )
        ).

printMessages(Delta,S) :- nl, write('  Program state= '), write(Delta), nl,
     write('  Current situation= '), write(S), nl.

sense(E,S):-
nl, write('>> Enter: an exogenous program or noop if none occurs.'),
           nl, read(E1),

 /* IF an exogenous is  noop,  or a terminating (hence, legal) Golog program
   THEN  this simulator will consider it as just happened.  */

( (E1 = noop; offline(E1,S,Sres)) -> E=E1 ;

 /* ELSE  print error message, and try again.  */

    write(">> Your program is not possible. Try again."), nl, sense(E,S)).
```

```
recover(M,N,PrOld,Exo,S,PrNew) :- M =< N, straightLineProgram(Plan,S,M),
          offline(Plan : PrOld, S, Sg),   /* Can we recover using Plan? */
          PrNew=(Plan : PrOld),      /* Yes: prefix old program by Plan */
          nl,
          write('  New program= '), write(PrNew),
          nl;
           /* Otherwise, try to find a longer sequence of actions */
          M < N,
           /* Complexity bound is not exceeded yet  */
          M_inc is M+1,
          recover(M_inc,N,PrOld,Exo,S,PrNew).

recover(M,N,PrOld,Exo,S,PrNew) :- M > N, nl,
                 write('-------------------------'),
                 write('|  Recovery procedure FAILED  |'),
                 write('-------------------------'),nl,
                  PrNew=nil.

straightLineProgram(Plan,S,K) :- K >= 1,
                  primitive_action(A), poss(A,S),
                  goodAction(Plan,A),
                    ( K =:= 1  -> Plan=A ;
                      K_dec is K -1,
                      straightLineProgram(TailPlan,do(A,S),K_dec),
                       Plan=(A : TailPlan)
                     ).

/* Elaborate in the future: A is a good action to perform after Plan if A
   does not undo the results of the last action in Plan. Otherwise, A is futile.
*/
goodAction(Plan,A).
```

Appendix 3: the blocks-world and the Golog program

```
/* This program has to build (non-deterministically) one of the two towers
                r              p
                o              a
                m      or      r
                e              i
                               s
given several blocks of sorts r, o, m, e, p, a, r, i. Final positions of other
blocks can be arbitrary.
```

```
*/

/*  Primitive Action Declarations  */

primitive_action(moveToTable(X)).
primitive_action(move(X,Y)).

/*  Action Precondition Axioms  */

poss(move(X,Y),S) :- clear(X,S), clear(Y,S), not X = Y.

poss(moveToTable(X),S) :- clear(X,S), not ontable(X,S).

/*  Successor State Axioms  */

on(X,Y,do(A,S)) :- A = move(X,Y) ;
                   on(X,Y,S), A \not= moveToTable(X), A \not= move(X,Z).

ontable(X,do(A,S)) :- A = moveToTable(X) ;
                      ontable(X,S), A \not= move(X,Y).

clear(X,do(A,S)) :- (A = move(Y,Z) ; A = moveToTable(Y)), on(Y,X,S) ;
                    clear(X,S), A \not= move(Y,X).

/* Initial Situation: everything is on table and clear.    */

r(r1).   r(r2).
o(o1).  o(o2).  o(o3).
m(m1). m(m2).
e(e1).  e(e2).

/* There is no block with the letter "p". */

a(a1).
i(i1).  i(i2).
s(s7).

ontable(r1,s0).  ontable(r2,s0).
ontable(o1,s0).  ontable(o2,s0).  ontable(o3,s0).
ontable(m1,s0).  ontable(m2,s0).
ontable(e1,s0).  ontable(e2,s0).
ontable(s7,s0).
ontable(a1,s0).
ontable(i1,s0).  ontable(i2,s0).
```

```
ontable(n,s0).
ontable(f,s0).

clear(r1,s0).  clear(r2,s0).
clear(o1,s0).  clear(o2,s0).  clear(o3,s0).
clear(m1,s0).  clear(m2,s0).
clear(e1,s0).  clear(e2,s0).
clear(s7,s0).
clear(a1,s0).
clear(i1,s0).  clear(i2,s0).
clear(n,s0).
clear(f,s0).

build :- online((tower : ?(goal)), s0, S).

proc(tower,
makeParis # makeRome).

proc(makeParis,
    pi(y5, ?(s(y5) & ontable(y5) & clear(y5)) :
        pi(y4, ?(i(y4)) : move(y4,y5) :
            pi(y3, ?(r(y3)) : move(y3,y4) :
                pi(y2, ?(a(y2)) : move(y2,y3) :
                    pi(y1, ?(p(y1)) : move(y1,y2))))))).

proc(makeRome,
    pi(x4, ?(e(x4) & ontable(x4) & clear(x4)) :
        pi(x3, ?(m(x3)) : move(x3,x4) :
            pi(x2, ?(o(x2)) : move(x2,x3) :
                pi(x1, ?(r(x1)) : move(x1,x2)))))).

goal(S) :-   p(Y1,S), a(Y2,S), r(Y3,S), i(Y4,S), s(Y5,S), ontable(Y5,S),
on(Y4,Y5,S), on(Y3,Y4,S), on(Y2,Y3,S), on(Y1,Y2,S), clear(Y1,S);
r(X1,S), o(X2,S), m(X3,S), e(X4,S),
ontable(X4,S), on(X3,X4,S), on(X2,X3,S), on(X1,X2,S), clear(X1,S).

  restoreSitArg(ontable(X),S,ontable(X,S)).
  restoreSitArg(on(X,Y),S,on(X,Y,S)).
  restoreSitArg(clear(X),S,clear(X,S)).
  restoreSitArg(goal,S,goal(S)).
```