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

KR'98, Trento, Italy, June 2 – June 5, 1998.

Introduction

High-level robot program: robot program formulated in terms of high-level representation of the world.

Atomic actions: associated to actual robot's commands. Commands activate complex low-level behaviors (e.g. $followCorridor$, $moveObjOnObj$).

Problem: when program **executed** in actual environment, discrepancy between real world and its representation may show up.

$\Longrightarrow$ Without **feedback** from environment:

$$Program + enviroment \longmapsto failure$$

Introduction (cont.)

How take into account feedback from environment?

Possible solutions:

- **Sensing controlled by program**

$$Complex\ program + enviroment \longmapsto success$$

  but requires inserting appropriate sensing actions every time possibility of discrepancy is foreseen.

- **Execution monitoring**

$$Program||Monitor + enviroment \longmapsto success$$

  Execution monitor senses environment asynchronously wrt program.

  - Program can remain simple, exploiting execution monitor for correct execution under exceptional circumstances.
  - Execution monitors can be designed once and for all, for supervising large classes of programs.

Introduction (cont.)

In this talk: a situation calculus account of **execution monitoring**.

For stressing role of execution monitor, we assume:

- program is sequential (no concurrent processes)
- program does not do sensing

Also, we assume:

- formalization of the world is correct
- discrepancies are sparse

Situation calculus

Situation calculus: Second order language for representing dynamically changing worlds [McCarthy63,McCarthy&Hayes69,Reiter91].

Language includes:

- Special symbols S_0, $do(a, s)$, $Poss(a, s)$
- Primitive actions $move(b_1, b_2)$, $moveToTable(b)$
- Fluents $On(b_1, b_2, s)$, $Clear(b, s)$ $OnTable(b, s)$
- Situation independent predicates $R(b)$, $O(b)$, $M(b)$, $E(b)$

Basic action theory [Reiter91,Reiter98]:

- Axioms describing initial situation S_0. *The description can be incomplete.*
- Action precondition axioms (one per action).

$$Poss(moveToTable(b), s) \;\equiv\; Clear(b) \wedge \neg OnTable(b, s)$$

- Successor state axioms (one per fluent). *These solve the frame problem.*

$$\begin{aligned} &OnTable(b, do(a, s)) \;\equiv \\ &\qquad a = moveToTable(b, s) \;\vee\; OnTable(b, s) \wedge \neg\exists b'.a = move(b, b') \end{aligned}$$

- Unique name assumption for actions.
- Some foundational domain independent axioms.

Golog

Golog [Levesque et al.97]: logic-based programming language. Primitive actions formalized in action theory. *GOLOG = alGOL in lOGic*

It includes following constructs:

- nil — empty program
- a — primitive action
- $\phi?$ — test condition
- $(\delta_1; \delta_2)$ — sequence
- $(\delta_1 \mid \delta_2)$ — nondeterministic choice of action
- $\pi v.\delta$ — nondeterministic choice of value
- δ^* — nondeterministic iteration
- $\mathbf{if}\,\phi\,\mathbf{then}\,\delta_1\,\mathbf{else}\,\delta_2$ — if-then-else
- $\mathbf{while}\,\phi\,\mathbf{do}\,\delta$ — while-loop
- $\mathbf{proc}\ P(\vec{v})\ \ \delta_P$ — procedure definition

In ϕ above the situations argument is suppressed.

Example: Golog program for dismounting all blocks towers

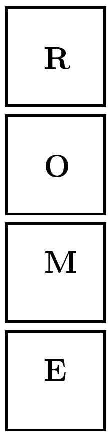
$$
\begin{array}{l}
\mathbf{while}\ \exists b, b'.On(b, b')\ \mathbf{do} \\
\{\quad \pi b.[Clear(b) \wedge \exists b'.On(b, b')]?; \\
\qquad moveToTable(b) \\
\}
\end{array}
$$

Golog (cont.)

Robot executing Golog program does **informed choices** at choice points to guarantee successful termination.

Example: Golog program for building blocks tower spelling “PARIS” or “ROME”.

$$makeParis \mid makeRome$$



proc $makeRome$
$\{ \quad \pi\, b_0.[E(b_0) \wedge OnTable(b_0) \wedge Clear(b_0)]?\,;$
$\quad \pi\, b_1.M(b_1)?\,; move(b_1, b_0)\,;$
$\quad\quad \pi\, b_2.O(b_2)?\,; move(b_2, b_1)\,;$
$\quad\quad\quad \pi\, b_3.R(b_3)?\,; move(b_3, b_2)$
$\}$

proc $makeParis$
$\{ \quad \pi\, b_0.[S(b_0) \wedge OnTable(b_0) \wedge Clear(b_0)]?\,;$
$\quad \pi\, b_1.I(b_1)?\,; move(b_1, b_0)\,;$
$\quad\quad \pi\, b_2.R(b_2)?\,; move(b_2, b_1)\,;$
$\quad\quad\quad \pi\, b_3.A(b_3)?\,; move(b_3, b_2)\,;$
$\quad\quad\quad\quad \pi\, b_4.P(b_4)?\,; move(b_4, b_3)$
$\}$

If only blocks for R, O, M, E, A, I, S available, robot does not consider building tower PARIS, it builds tower ROME.

Golog's computation semantics

Semantics of Golog programs (in absence of external disturbance) is captured by relation

$$Do(\delta, s, s')$$

which selects all situations s' resulting from a **successfully terminating execution** of program δ starting from s.

In [Levesque et al.97] it is shown that for any program δ relation $Do(\delta, s, s')$ can be inductively defined in terms of a pure situation calculus formula not mentioning δ.

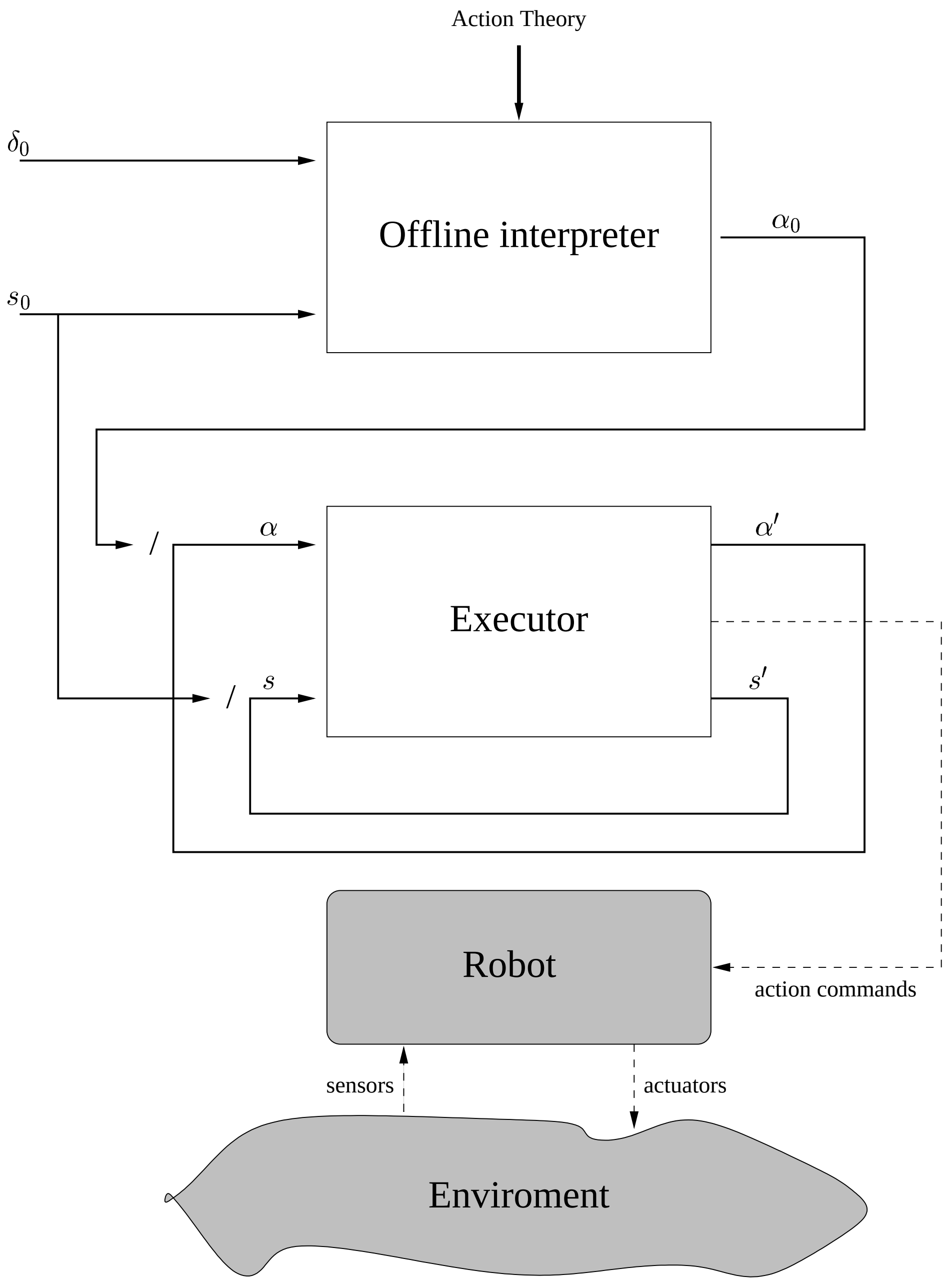
$$\begin{array}{rcl}
Do(nil, s, s') & \doteq & s' = s \\
Do(a, s, s') & \doteq & Poss(a[s], s) \wedge s' = do(a[s], s) \\
Do(\phi, s, s') & \doteq & \phi[s] \wedge s' = s \\
Do(\delta_1; \delta_2, s, s') & \doteq & \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s') \\
Do(\pi v.\delta, s, s') & \doteq & \exists v.Do(\delta, s, s') \\
\ldots & &
\end{array}$$

Offline Golog interpreter

From $Do(\delta, s, s')$, one can immediately lift an **offline** Golog interpreter.

- Interpreter computes an s' such that $Do(\delta, s, s')$ holds.
- From $s' = do(a_n, do(\ldots, do(a_1, s)\ldots))$, sequence of actions $a_1; \ldots; a_n$ to be executed in real world is extracted.

Offline-interpreter-based architecture



From open loop to closed loop

How can feedback from environment (as observed through robot's sensors) be considered?

Execution monitoring approach:

1. **Online interpreter** [DeGiacomo&Levesque98]: put ability of making reasoned choices (typical of offline interpreter) into executor. In this way program can react to external stimuli.

2. **Execution monitor**: monitor execution of program in environment, passing appropriate stimuli from environment to program, and possibly dealing with exceptional unexpected circumstances.

Golog's transition semantics

An alternative way to formulate semantics of Golog programs is in terms of **single steps** (either actions or tests). Captured by two relations [IJCAI97]:

- $Trans(\delta, s, \delta', s')$ – program δ executed in situation s may perform an action resulting in situation s' with program δ' remaining to be executed.
- $Final(\delta, s)$ – program δ may be considered successfully terminated in situation s.

$Do(\delta, s, s')$ can be re-expressed using reflexive-transitive closure of $Trans$ – "single step the program until you can successfully terminate".

Characterizing $Trans$ & $Final$ in situation calculus requires introducing new sort for *programs*.

$$
\begin{aligned}
&Trans(nil, s, \delta', s') \equiv False\\
&Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s)\\
&Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s\\
&Trans(\delta_1;\delta_2, s, \delta', s') \equiv \exists\gamma.\, Trans(\delta_1, s, \gamma, s') \wedge \delta' = \gamma;\delta_2 \;\vee\\
&\qquad\qquad Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')\\
&Trans(\pi v.\delta, s, \delta', s') \equiv \exists x.\, Trans(\delta^v_x, s, \delta', s')\\
&\ldots\\
&Final(nil, s) \equiv True\\
&Final(a, s) \equiv False\\
&Final(\phi?, s) \equiv False\\
&Final(\delta_1;\delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)\\
&\ldots\\
&Final(\delta^*, s) \equiv True\\
&\ldots
\end{aligned}
$$

Online Golog interpreter

From $Trans$ & $Final$, one can lift an **online** Golog interpreter:

- At each step interpreter computes a (δ', s') such that $Trans(\delta, s, \delta', s')$ holds.
- From $s' = do(a, s)$ action a to be executed in real world is extracted.
- Interpreter stops when $Final(\delta, s)$ holds.

Note:

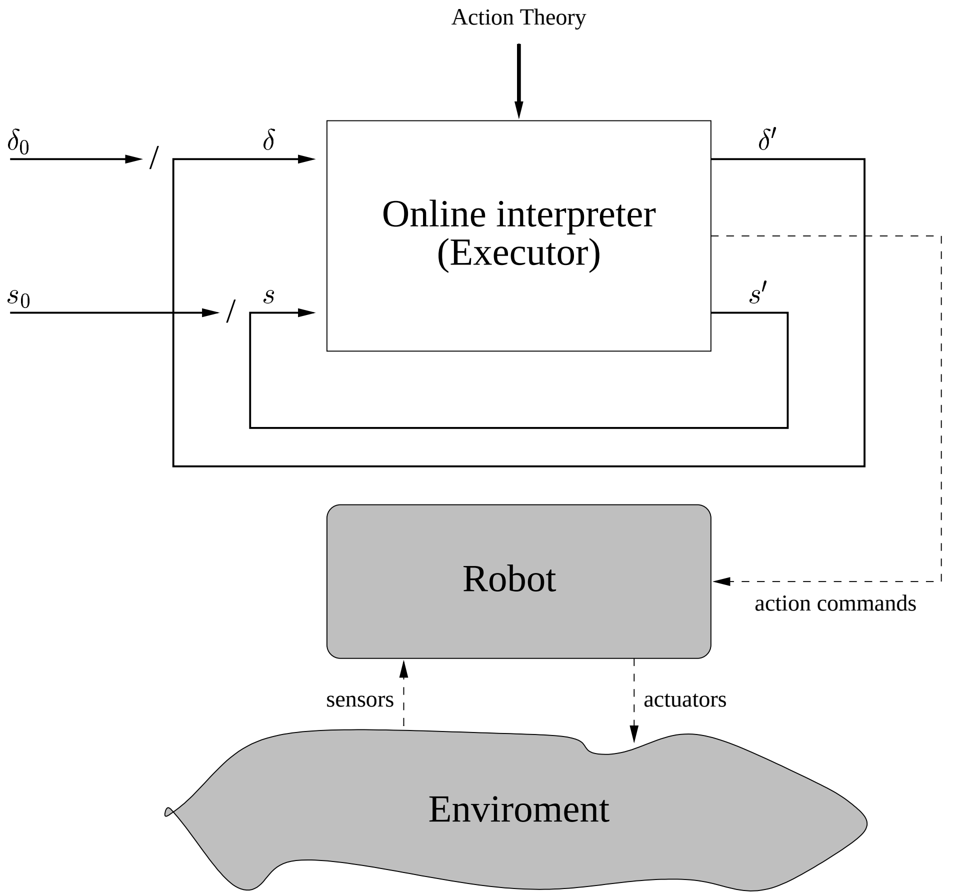
- Interpreter **cannot backtrack**, because it does actions in real world.
- Interpreter **can reason** to choose transitions to execute.

  For example:

  – **Brave online interpreter:** randomly choose any transition.

  – **Cautious online interpreter:** choose transition only if from resulting configuration (δ', s') program can successfully terminate, i.e. $\exists s_f . Do(\delta', s', s_f)$.

Online-interpreter-based architecture



Online Golog interpreter with execution monitor

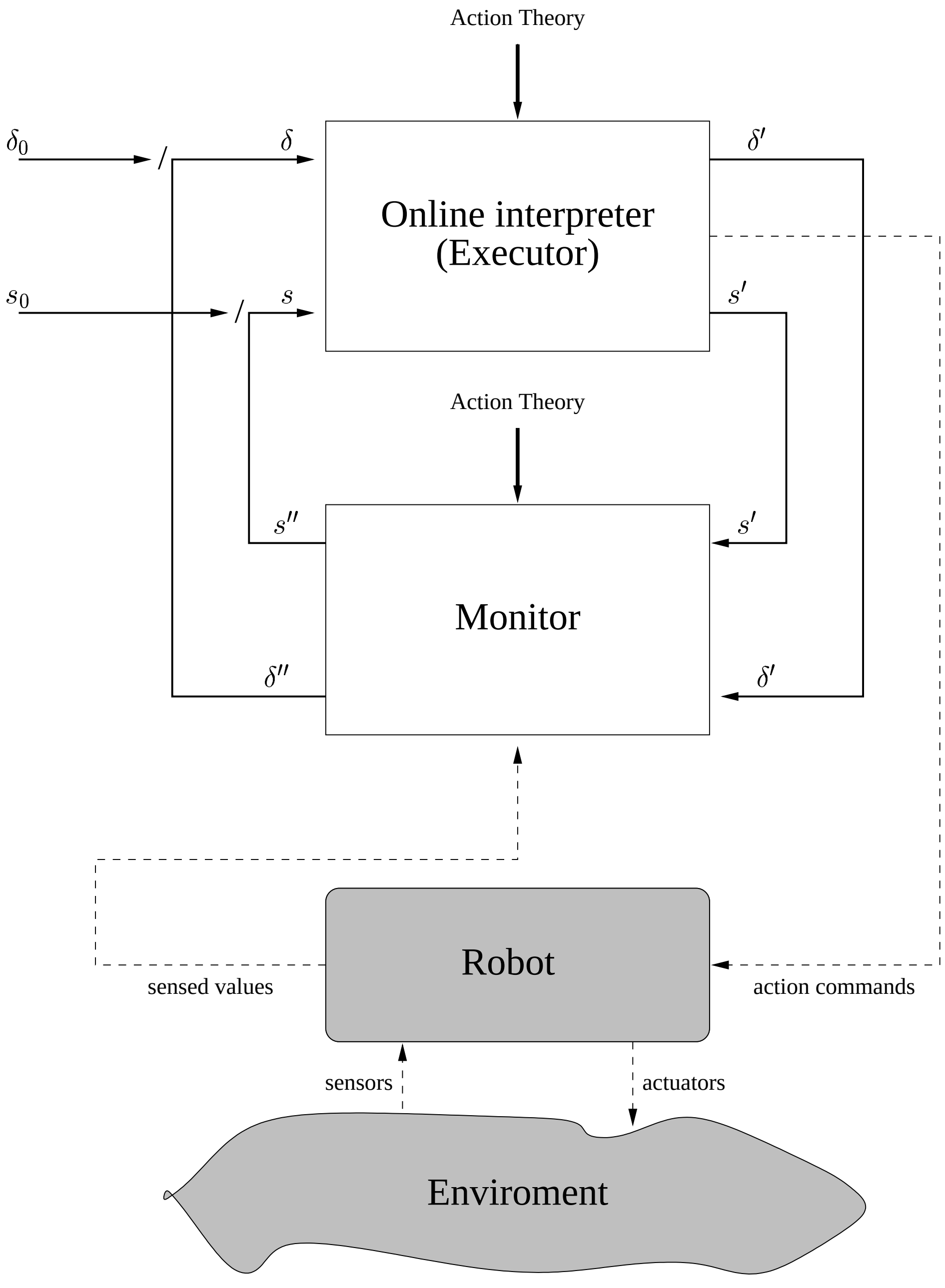
One gets closed loop architecture by cascading **online interpreter** and **execution monitor**.

Execution monitor responsible for:

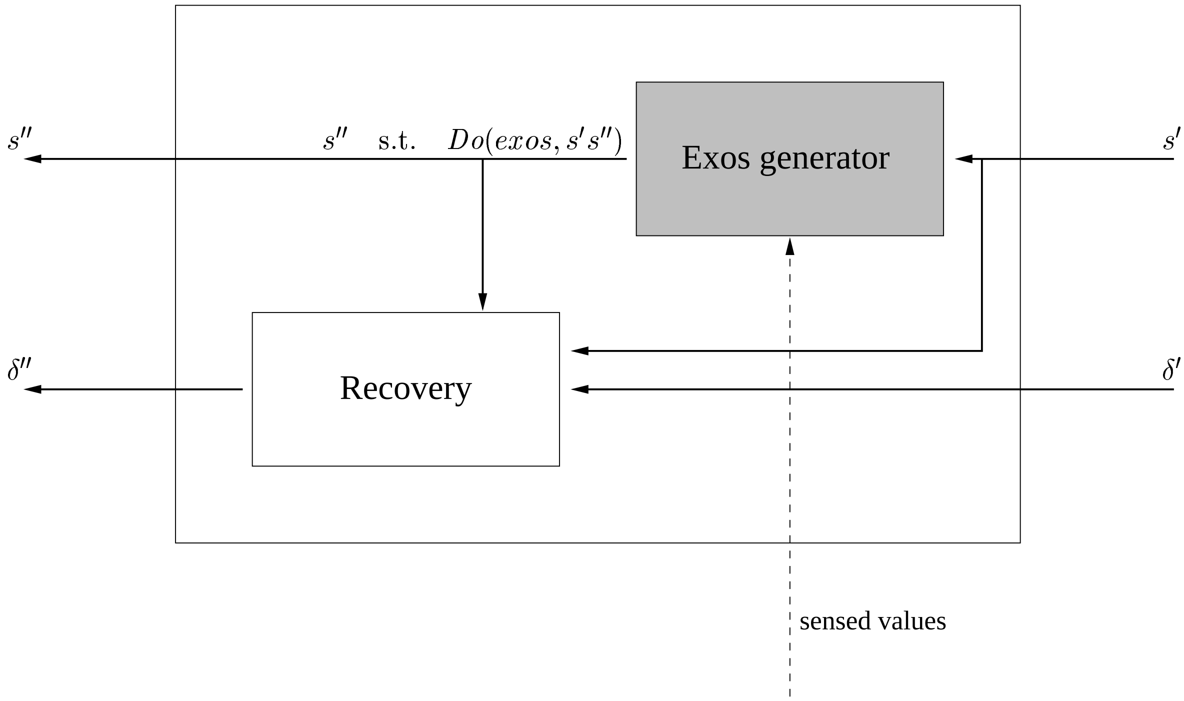
- Updating current situation using view of real world coming out of robot's sensors, i.e. *interfacing low-level sensing processes with high-level representation.*

- Possibly recovering form perceived discrepancies.

Execution monitoring architecture



Execution monitor



Execution monitor:

1. using robot's sensors, checks for **discrepancies** between state of the world as described by current situation and actual one

2. when discrepancy detected, generates one or more **exogenous events** to align current situation with actual state of the world

3. possibly modifies current program to **recover** from discrepancy.

Execution monitor formalization

Formal view of execution monitor:

$$
\begin{array}{ll}
Monitor(\delta', s', \delta'', s'') \equiv & \\
\quad \exists exos.Do(exos, s', s'') \;\wedge & \text{align current situation} \\
\quad Recovery(\delta', s', s'', \delta'') & \text{recover from discrepancy}
\end{array}
$$

- $exos$: program (sequence) formed by exogenous events.
  - Process generating program $exos$ is considered **extra-logical**.
  - However **exogenous events** forming $exos$ are **formalized** in theory of actions:
    - *Preconditions* reflect context in which given discrepancy can be detected.
    - *Effects* are used to align current situation to actual state of the world.

- $Recovery$ checks for **relevance** of discrepancy wrt configuration (δ', s'). If discrepancy relevant, a **recovery procedure** is called.

$$
\begin{array}{l}
Recovery(\delta', s', s'', \delta'') \doteq \\
\quad \neg Relevant(\delta', s', s'') \wedge \delta'' = \delta' \quad \vee \\
\quad\; Relevant(\delta', s', s'') \wedge Recover(\delta', s', s'', \delta'')
\end{array}
$$

Execution monitor (cont.)

Varying $Relevant$ and $Recover$ one gets a wide variety of monitors, differing in

- range of applicability
- automated reasoning requirements

Key point: find good trade-off between generality and implementability.

A specific execution monitor

Assumption: programmer provides program δ_0. No analysis of program is done.

Relevant: discrepancies that change state (i.e. values of fluents) of current situation:

$$Relevant(\delta', s', s'') \equiv \neg SameState(s', s'')$$

Recover: determine sequential plan p (a straight-line program) that leads from s'' to a situation s_r equivalent to s' (i.e. $SameState(s', s_r)$), and returns $\delta'' = p\,;\delta'$.

$$\begin{aligned}
&Recover(\delta', s', s'', \delta'') \equiv \\
&\quad \exists p.\ straightLineProg(p) \ \wedge\ \exists s_r.Do(p, s'', s_r) \ \wedge\ SameState(s', s_r) \ \wedge \\
&\qquad \delta'' = p\,;\delta'
\end{aligned}$$

Another specific execution monitor

(Implemented)

Assumption: programmer provides program δ_p and postcondition $goal$ (a first order sentence describing the "goal" of δ_p). Program to execute:

$$\delta_0 \; = \; \delta_p; goal?$$

Relevant: discrepancies that prevent successful termination of current program δ':

$$Relevant(\delta', s', s'') \equiv \neg \exists s_f . Do(\delta', s'', s_f)$$

Recover: determine shortest straight-line program that prefixed to δ' yield to successful termination:

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

Implementation

From logical formalization a Prolog implementation can be easily extracted *(accepting Prolog closed world assumption – see [Reiter98])*:

```
onlineEM(Prog,S) :-

  final(Prog,S) ;

  trans(Prog,S,Prog1,S1), do(Prog1,S1,Sf), !,
  outputAction(S,S1),
  monitor(Prog1,S1,Prog2,S2),
  onlineEM(Prog2,S2).



monitor(Prog1,S1,Prog2,S2) :-
  generateExos(S1,S2),
  recovery(Prog1,S1,S2,Prog2), !.
```

where, for example:

- `outputAction` outputs current action command (deterministic).
- `generateExos` inputs $exos$ and produces aligned situation accordingly (deterministic).
- `recovery` implements recovery strategy in previous slide.

Example: blocks world

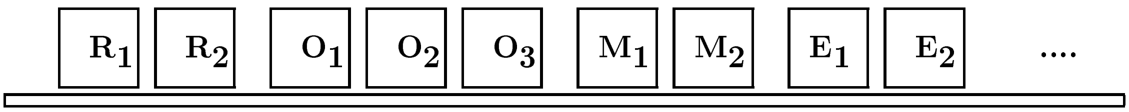
(Implemented)

- Action theory for blocks world.
- Program to execute: $\delta_0 \ = \ \delta_p; goal?$
  - Program $\delta_p \ = \ makeParis \mid makeRome$.
  - Postcondition $goal(s) \ = \ spellsParis(s) \lor spellsRome(s)$

$$\begin{aligned} spellsRome(s) \ \doteq \ \exists b_0, b_1, b_2, b_3.\ & E(b_0) \land OnTable(b_0, s) \land \\ & M(b_1) \land On(b_1, b_0, s) \land \\ & O(b_2) \land On(b_2, b_1, s) \land \\ & R(b_3) \land On(b_3, b_2, s) \land Clear(b_3, s) \end{aligned}$$

$$spellsParis(s) \ \doteq \ \ldots$$

- Initial situation:



Other blocks are present, e.g. a block with letter "N", one with letter "F", but no block with letter "P".

Example: 1st execution step

Because no "P" block, procedure $makeParis$ cannot be successfully completed. Cautious online interpreter selects procedure $makeRome$.

First execution step:

$$\pi\, b_0.[E(b_0) \wedge OnTable(b_0) \wedge Clear(b_0)]?\,;$$

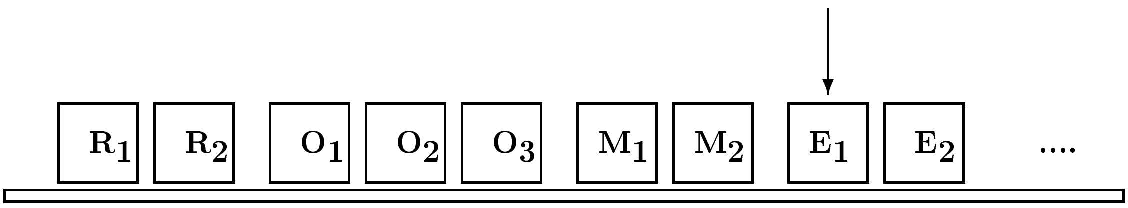
determines block E_1 as base of tower ROME.

Current program δ':

nil ;
$\pi\, b_1.M(b_1)?\,; move(b_1, E_1)\,;$
$\quad\pi\, b_2.O(b_2)?\,; move(b_2, b_1)\,;$
$\quad\quad\pi\, b_3.R(b_3)?\,; move(b_3, b_2)$;
$goal?$

where nil results after first step.

Current situation $s'(= s)$:



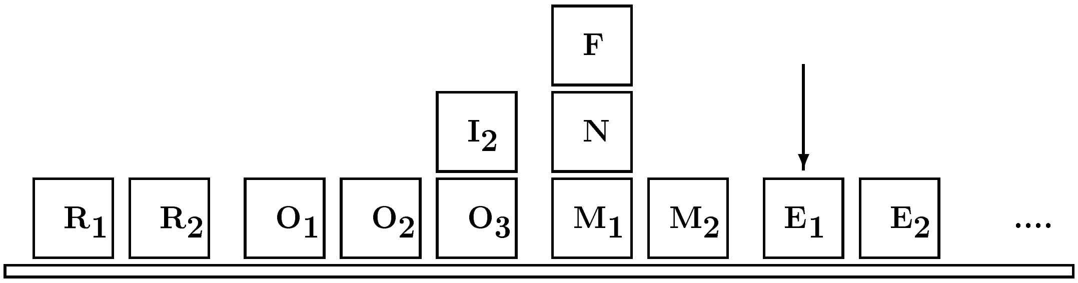
Example: 1st discrepancy

A discrepancy is detected. Sequence of exogenous events:

$$move(N, M_1); move(F, N); move(I_2, O_3)$$

$\Longrightarrow$ aligned situation s'':

$$do(move(I_2, O_3), do(move(F, N), do(move(N, M_1), S_0)))$$



Since program δ'

nil ;
$\pi\, b_1.M(b_1)?\,; move(b_1, E_1)\,;$
$\quad \pi\, b_2.O(b_2)?\,; move(b_2, b_1)\,;$
$\quad\quad \pi\, b_3.R(b_3)?\,; move(b_3, b_2)$;
$goal?$

can still be successfully completed (pick M_2, move it on top of E_1, etc.) $\Longrightarrow$ no recovery is necessary.

Example: 2nd execution step & 2nd discrepancy

A further step is performed.

Current program δ' is now:

$nil\,;$
$\quad move(M_2, E_1)\,;$
$\quad\quad \pi\, b_2.O(b_2)?\,; move(b_2, M_2)\,;$
$\quad\quad\quad \pi\, b_3.R(b_3)?\,; move(b_3, b_2)\,;$
$goal?$

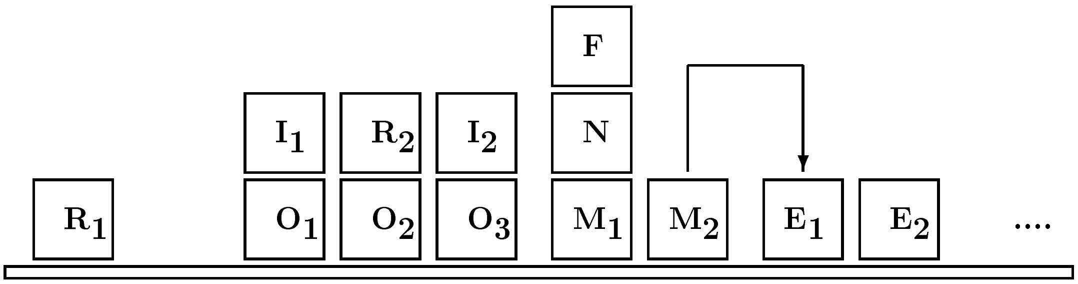
Current situation s': $do(move(I_2, O_3), do(move(F, N), do(move(N, M_1), S_0)))$

Another discrepancy is detected. Sequence of exogenous events:

$$move(I_1, O_1); move(R_2, O_2)$$

$\Longrightarrow$ aligned situation s'':

$$do(move(R_2, O_2), do(move(I_1, O_1),\\ do(move(I_2, O_3), do(move(F, N), do(move(N, M_1), S_0)))))$$

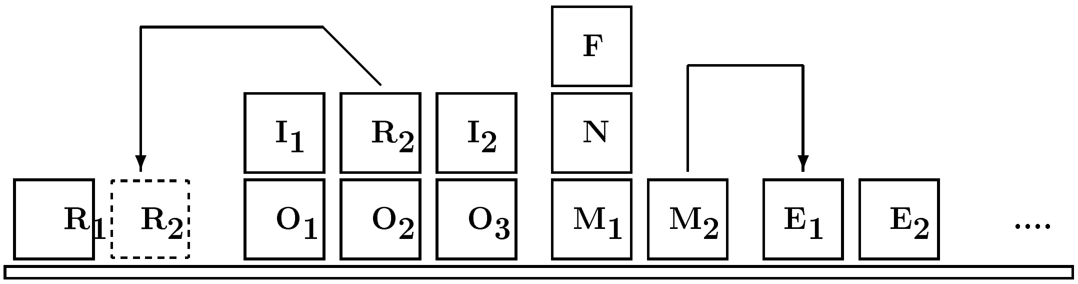


Example: recovery

Current program δ' cannot be completed $\Longrightarrow$ discrepancy is **relevant** $\Longrightarrow$ recovery procedure is called.

Recovery procedure (a **planner**) finds a minimal sequence of actions that prefixed to program δ' results in successful termination. In this case simply:

$$moveToTable(R_2)$$



So current program is now δ'':

$moveToTable(R_2)\,;$
 $nil\,;$
  $move(M_2, E_1)\,;$
   $\pi\, b_2.O(b_2)?\,; move(b_2, M_2)\,;$
    $\pi\, b_3.R(b_3)?\,; move(b_3, b_2)\,;$
$goal?$

⋮
Etc.

Discussion

We have given a logical account in the situation calculus of execution monitoring.

Having based our fomalization on logic buy us several benefits:

- Precisely formulate requirements and properties for monitors and monitored programs.
- Formally verify whether specific monitor satisfies requirements.
- Assess ranges of applicability of recovery strategies.

Prolog implementation can be directly lifted from logical formalization.

Ongoing and future work

- Devise suitable notions of relevance and recovery with good trade-off between generality and implementability.
- Explore correspondence with controllable systems in discrete event control theory.
- Get time into the picture, for example, monitoring programs written in temporal Golog.
- Implement these ideas on actual robots (e.g. ongoing work on the RWI B21 autonomous robot at the University of Toronto).