Integrating Planning into Reactive High-Level Robot Programs

Yves Lespérance and **Ho-Kong Ng**[1]

Abstract. IndiGolog is a high-level programming language for robots and intelligent agents that supports on-line planning and plan execution in dynamic and incompletely known environments. Programs may perform sensing actions that acquire information at run-time and react to exogenous actions. In this paper, we show how IndiGolog can be used to write robot control programs that combine planning, sensing, and reactivity. Moreover, we present enhancements to IndiGolog in three areas: a more effective replanning mechanism for situations where the environment has changed, an approach to planning in dynamic settings that uses a simulated environment, and a mechanism that allows planning to be done within a larger program that includes reactive threads.

1 Introduction

Synthesizing plans at run-time provides great flexibility, but is often computationally infeasible. In [9], it is argued that *high-level program execution* is a more practical alternative. The idea is that instead of searching for a sequence of actions that takes the agent from an initial state to some goal state as in planning, the task is to find a sequence of actions that constitutes a legal execution of some high-level program. By *high-level program*, we mean one whose primitive instructions are domain-dependent actions of the agent, whose tests involve domain-dependent predicates that are affected by the actions, and whose code may contain nondeterministic choice points where lookahead is necessary to make a choice that leads to successful termination. As in planning, to find a sequence that constitutes a legal execution of a high-level program, one must *reason* about the preconditions and effects of the actions within the program. However, if the program is almost deterministic, very little searching is required; as more nondeterminism is included, the search task begins to resemble traditional planning.

In [9], *Golog* was proposed as a suitable language for expressing high-level programs for robots and autonomous agents. Golog supports program structures such as sequence, conditionals, loops, and non-deterministic choice of actions and arguments. It uses a situation calculus theory of action for the domain of the application to perform the reasoning required in executing the program. In [2, 3], an extension called *GonGolog* was introduced, adding support for concurrent processes with possibly different priorities, interrupts, and exogenous events. These new constructs are useful for writing controllers that react to environmental events while working on certain tasks. Both languages have been implemented in Prolog.

Golog was used to design a high-level robot controller for a mail delivery application [13]. Later, an enhanced version of that controller that can react to new shipment orders and navigation failures was implemented in ConGolog and tested on a RWI-B12 and Nomad Super Scout [7]. A team at the University of Bonn also used Golog to control a very successful museum guide robot [1].

Key features of real-world robot control applications are that the environment is dynamic and that the system has incomplete knowledge and must acquire information at run-time by performing sensing actions. However, Golog and ConGolog, like earlier planning-based systems, assume an off-line search model. That is, the interpreter is taken to search all the way to a final state of a program before any action is really executed. This can be a serious problem if, for instance, the program involves a long running application, or if part of the program depends on information that can only be obtained by doing sensing at run-time. It is also impractical to spend large amounts of time searching for a complete plan when the environment is very dynamic. To address this limitation, an extension of ConGolog called *IndiGolog* [4] has been developed; it supports the inclusion of planning/search components within an overall deterministic program that is to be executed *incrementally* in conjunction with sensing of the environment.

Although IndiGolog as defined in [4] provides many useful features for designing controllers that operate in dynamic and incompletely known environments, it still has some limitations. Moreover, it has never been used to implement a non-trivial robotics application. In this paper, we show how robot control programs that exploit IndiGolog's new features can be written. Moreover, we present enhancements to the IndiGolog language and its interpreter in three areas: a more effective replanning mechanism for situations where the environment has changed, an approach to planning in dynamic settings that uses a simulated environment, and a mechanism that allows planning to be performed within a larger program that includes reactive threads. In the next section, we give an overview of IndiGolog. Following that, we discuss the problems that motivated our enhancements to IndiGolog, the solutions that were developed, and example robot control programs that use these.

2 IndiGolog

As mentioned, our high-level programs contain primitive actions and tests of predicates that are domain-dependent, and an interpreter for such programs must reason about these. We specify the required domain theories in the situation calculus [10], a language of predicate logic for representing dynamically changing worlds. In this language, 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 and the term $do(\alpha, s)$ denotes the situation resulting from action α being performed in situation s.

[1] Department of Computer Science, York University, Toronto, ON, Canada M3J 1P3, email: {lesperan,hokong}@cs.yorku.ca

Relations that vary from situation to situation, called *predicate fluents*, are represented by predicate symbols that take a situation term as last argument; for example, $Holding(o, s)$ might mean that the robot is holding object o in situation s. Similarly, functions whose value varies with the situation, *functional fluents*, are represented by function symbols that take a situation argument. The special predicate $Poss(\alpha, s)$ is used to represent the fact that primitive action α is executable in situation s. A domain of application will be specified by theory that includes the following types of axioms:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action α, which characterizes $Poss(\alpha, s)$.
- Successor state axioms, one for each fluent F, which characterize the conditions under which $F(\vec{x}, do(a, s))$ holds in terms of what holds in situation s; these axioms may be compiled from effects axioms, but provide a solution to the frame problem [12].
- Sensed fluent axioms, which relate the value returned by a sensing action to the fluent condition it senses in the environment (since these are not used in this paper, we omit the details; see [4]).
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

An IndiGolog program implementing an agent or robot controller will include such a domain theory.

An IndiGolog program also includes a procedural part that specifies the behavior of the agent or robot. This is specified using the following constructs:

α,	primitive action
$\phi?$,	wait for a condition[2]
$(\delta_1; \delta_2)$,	sequence
if ϕ **then** δ_1 **else** δ_2 **endIf**,	conditional
while ϕ **do** δ **endWhile**,	loop
proc $\beta(\vec{x})$ δ **endProc**,	procedure definition
$\beta(\vec{t})$,	procedure call
$(\delta_1 \mid \delta_2)$,	nondeterministic choice between actions
$(\pi\ \vec{x})[\delta]$,	nondeterministic choice of arguments
δ^*,	nondeterministic iteration
$(\delta_1 \parallel \delta_2)$,	concurrent execution
$(\delta_1 \rangle\!\rangle\ \delta_2)$,	concurrency with different priorities
$\delta^{\parallel}$,	concurrent iteration
$< \vec{x} : \phi \rightarrow \delta >$,	interrupt
search(δ),	search block

The nondeterministic constructs include $(\delta_1 \mid \delta_2)$, which nondeterministically choses between programs δ_1 and δ_2, $(\pi\vec{x})[\delta]$, which nondeterministically picks a binding for the variables $\vec{x}$ and performs the program δ for this binding of $\vec{x}$, and δ^*, which means performing δ zero or more times. Concurrent processes are modeled as interleavings of the primitive actions involved. A process may become blocked when it reaches a primitive action whose preconditions are false or a wait action $\phi?$ whose condition ϕ is false. Then, execution of the program may continue provided another process executes next. In $(\delta_1 \rangle\!\rangle\ \delta_2)$, δ_1 has higher priority than δ_2, and δ_2 may only execute when δ_1 is done or blocked. $\delta^{\parallel}$ is like nondeterministic iteration δ^*, but the instances of δ are executed concurrently rather than in sequence. Finally, an interrupt $< \vec{x} : \phi \rightarrow \delta >$ has variables $\vec{x}$, a trigger condition ϕ, and a body δ. If the interrupt gets control from higher priority processes and the condition ϕ is true for some binding of the variables, the interrupt triggers and the body is executed with the variables taking these values. Once the body completes execution, the interrupt may trigger again. We discuss the search block construct below and give example IndiGolog programs in the next sections.

A prototype IndiGolog interpreter has been implemented in Prolog. This implementation requires that the program's domain theory be expressible as Prolog clauses, essentially closed theories; note that this is a limitation of this particular implementation, not the framework.

IndiGolog's semantics is defined in terms of *transitions*, in the style of structural operational semantics [11, 6]. A transition is a single step of computation, either a primitive action or testing whether a condition holds in the current situation. Two special predicates are introduced, $Final$ and $Trans$, where $Final(\delta, s)$ is intended to say that program δ may legally terminate in situation s, and where $Trans(\delta, s, \delta', s')$ is intended to say that program δ in situation s may legally execute one step, ending in situation s' with program δ' remaining. $Trans$ and $Final$ are characterized by axioms such as:

$$Trans(\alpha, s, \delta, s') \equiv \quad \text{primitive action}$$
$$Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s)$$

$$Final(\alpha, s) \equiv False$$

$$Trans([\delta_1; \delta_2], s, \delta, s') \equiv \quad \text{sequence}$$
$$Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta, s')$$
$$\vee\ \exists \delta'. \delta = (\delta'; \delta_2) \wedge Trans(\delta_1, s, \delta', s')$$

$$Final([\delta_1; \delta_2], s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

The first axiom says that a program involving a primitive action a may perform a transition in situation s provided that a is possible in s, with the resulting situation being $do(a, s)$ and the remaining program being the empty program nil. The second axiom says that a program with a primitive action remaining can never be considered to have terminated. The third axiom says that one can perform a transition for a sequence by performing a transition for the first part, or by performing a transition for the second part provided that the first part has already terminated. The last axiom says that a sequence has terminated when both parts have terminated.

IndiGolog has a search block construct $\mathbf{search}(\delta)$ for doing planning in a program ([4] uses the symbol Σ for $\mathbf{search}$). In executing $\mathbf{search}(\delta)$, the interpreter searches to find a sequence of transitions (primitive actions or tests) that leads to a final situation for the block.[3] For example, for the program $\mathbf{search}(a_1; False? \mid a_2; a_3)$, where $\mid$ denotes nondeterministic choice between the two argument programs, the interpreter would avoid the left branch which does not lead to a final situation and chose to perform primitive action a_2 (assuming preconditions are satisfied). In [4], the semantics of search blocks is defined by the following axioms:

$$Trans(\mathbf{search}(\delta), s, \delta', s') \equiv$$
$$(\exists \gamma'). \delta' = \mathbf{search}(\gamma') \wedge Trans(\delta, s, \gamma', s') \wedge$$
$$(\exists \gamma'', s''). Trans^*(\gamma', s', \gamma'', s'') \wedge Final(\gamma'', s''),$$
$$Final(\mathbf{search}(\delta), s) \equiv Final(\delta, s).$$

The first axiom says that one can make a transition from a program $\mathbf{search}(\delta)$ and a situation s to a program $\mathbf{search}(\gamma')$ and situation s', provided that one can make a transition from δ and s to γ' and s', and there is a sequence of transitions that leads from these to a con-

[2] Here, ϕ stands for a situation calculus formula with all situation arguments replaced by now; $\phi[s]$ will denote the formula obtained by substituting the situation term s for occurrences of now in all fluents appearing in ϕ. For simplicity, we often leave out leave out the situation argument now altogether.

[3] When executing nondeterministic code that is not in a search block, IndiGolog can choose transitions arbitrarily.

figuration that is $Final$, i.e., where the program may legally terminate; $Trans^*$ is the reflexive transitive closure of $Trans$. The second axiom simply says that a search block can be considered to have successfully terminated in a situation s if and only the program that remains in the block can successfully terminate in s. For more detail on the IndiGolog language and semantics, see [4], as well as the references on ConGolog [2, 3].

3 Enhancing the Replanning Mechanism

As explained, in executing a search block, IndiGolog performs lookahead; it searches for an execution path (sequence of transitions) that successfully gets it to the end of the search block, and then executes the path. For efficiency, the interpreter caches the path found during the initial search and keeps following it as long as no exogenous action occurs. IndiGolog monitors for exogenous actions when the robot/agent is operating in a dynamic environment — a form of sensing. So when an exogenous action does occur, the interpreter rechecks the path and if it no longer leads to a final situation, a new search is performed. Following the semantics given above, the interpreter in [4] performs this replanning search for the program that is currently left to execute in the search block. In effect, the interpreter has committed to finding an execution that follows the path through the program taken so far. In some cases, there may be no such executions. But this is unnecessarily restrictive. The only thing the agent is really committed to are the actions performed so far. It should be able to take another path through the program provided it involves performing the same initial sequence of actions. For example, if the initial program is $\mathbf{search}(a_1; a_2 \mid a_1; a_3)$ and the initial search leads to selecting the first branch and performing the action a_1, and then an exogenous action occurs that makes a_2 not executable, the interpreter will be stuck with $\mathbf{search}(a_2)$ as remaining program and fail; it should realize that a_3 is a possible next action that is on a path through the original program that starts with the actions it has already performed.

We have modified the IndiGolog interpreter so that the initial program and situation of the search block are memorized, and when replanning is performed, the search starts from this initial program and situation and goes through the current situation. The semantics for this is as follows:

$$Trans(\mathbf{search}(\delta), s, \delta', s') \equiv Trans(\mathbf{search}'(\delta, \delta, s), s, \delta', s'),$$

$$\begin{aligned} &Trans(\mathbf{search}'(\delta, \delta_i, s_i), s, \delta', s') \equiv \\ &\quad (\exists \gamma, \gamma').\delta' = \mathbf{search}'(\gamma', \delta_i, s_i) \wedge \\ &\qquad Trans^*([\delta_i \parallel \delta_{exo}], s_i, [\gamma \parallel \delta_{exo}], s) \wedge \\ &\qquad Trans(\gamma, s, \gamma', s') \wedge \\ &\qquad (\exists \gamma'', s'').Trans^*(\gamma', s', \gamma'', s'') \wedge Final(\gamma'', s''), \end{aligned}$$

$$Final(\mathbf{search}(\delta), s) \equiv Final(\mathbf{search}'(\delta, \delta, s), s),$$

$$\begin{aligned} &Final(\mathbf{search}'(\delta, \delta_i, s_i), s,) \equiv \\ &\quad (\exists \gamma).Trans^*([\delta_i \parallel \delta_{exo}], s_i, [\gamma \parallel \delta_{exo}], s) \wedge Final(\gamma, s), \end{aligned}$$

where $\delta_{exo} \stackrel{\text{def}}{=} ((\pi\, a)[Exo(a)?; a])^*$.

In the first and third axioms, we rewrite $\mathbf{search}(\delta)$ into $\mathbf{search}'(\delta, \delta, s)$ to memorize the the initial search block program δ and initial situation s. In the second axiom that defines $Trans$, in the path that leads from the initial situation s_i to the current situation s, we run the initial search block program δ_i concurrently with a program δ_{exo} that can generate the exogenous actions that have occurred. The next transition can be on any path through the original program that involves the non-exogenous actions that have already been done (and leads to a final situation). Similarly in the last axiom defining $Final$, we allow termination if the current situation is final for any path through the original program that involves the non-exogenous actions that have already been done.

Let us present an example of a useful robotics program that exploits this enhancement. The program controls a robot to serve some clients, for example, picking up and delivering mail, choosing the order in which the clients are served so as to minimize the distance traveled by the robot:

proc $minimizeDistance(distance)$
  $serveAllClientsWithin(distance)$
  | % or
  $minimizeDistance(distance + 1)$
endProc

proc $serveAllClientsWithin(distance)$
  $\neg(\exists c)\ ClientToServe(c)?$ $\quad \%\ if\ no\ clients\ to\ serve,\ done$
  | % or
  $(\pi c, d)[(ClientToServe(c) \wedge$ $\quad \%\ choose\ a\ client\ to\ serve$
    $d = distanceTo(c) \wedge d \leq distance)?;$
    $goto(c);$
    $serveClient(c);$
    $serveAllClientsWithin(distance - d)]$
endProc

The program is run by executing $\mathbf{search}(minimizeDistance(0))$. The $mimimizeDistance$ procedure does an iterative deepening search to find a path that serves all the clients and minimizes the distance traveled. Thus initially, in $minimizeDistance(0)$, the interpreter can choose between executing $serveAllClientsWithin(0)$ or $minimizeDistance(1)$, and looks for one that leads to a final situation. Suppose that the robot has 3 clients to serve initially and has to travel a minimum of 6 units of distance to serve them all. Since it is impossible to serve everyone in 0 distance (i.e., $serveAllClientsWithin(0)$ fails), $minimizeDistance(1)$ is chosen. Then, the interpreter faces another choice between $serveAllClientsWithin(1)$ and $minimizeDistance(2)$. The distance bound will be incremented until it reaches 6 and $serveAllClientsWithin(6)$ succeeds. Then, the original interpreter commits to the program branch $serveAllClientsWithin(6)$ and starts executing it and delivering the mail. Now suppose that that at some later point, the robot receives another service request from a client — an exogenous action. It then has to find a new plan in order to serve all the clients including the new one. However, with the original IndiGolog semantics and interpreter, it is impossible to find such a plan since the distance bound cannot be increased from 6 within the advanced program $serveAllClientsWithin(6)$.

With our modified semantics and enhanced interpreter that redoes the search from the original program and situation, the bound can be incremented and a new minimum path that includes the actions already performed and serves all the clients can be found. The enhanced interpreter has been tested on a version of this mail delivery example, as well as others. Its Prolog implementation will be described in section 6.

4 Planning Using a Simulated Environment

In some cases, the robot/agent must not only react to exogenous actions, but also rely on their occurrence. For example, in a more realistic account of navigation, the controller may need to wait for an

exogenous action notifying it of whether the attempt to navigate to the client's location was successful. The $serveAllClientsWithin$ procedure of the previous section can be changed to do this as follows:

proc $serveAllClientsWithin(distance)$
  $\neg(\exists c)\ ClientToServe(c)?$
  $|$
  $(\pi c, d)[(ClientToServe(c)? \wedge$
    $d = distanceTo(c) \wedge d \leq distance)?;$
    $startGoto(c);$ $\%\ start\ to\ navigate\ to\ client$
    $robotState \neq Moving?;$ $\%\ wait\ until\ robot\ stops$
    if $robotState = Reached$ **then**
      $serveClient(c);$
    else
      $handleServiceFailure(c);$
    endIf;
    $serveAllClientsWithin(distance - d)]$
endProc

Here, the program performs the primitive action $startGoto(c)$ to initiate motion to client c's location, and then blocks at the test $robotState \neq Moving?$ until the robot stops moving. If the robot is successful in reaching the destination, the exogenous action $reachDest$ occurs (a signal from the lower-levels of the architecture), which changes $robotState$ to $Reached$. Then, the robot can deliver the mail to the client. If instead the robot fails to reach the destination, then the exogenous action $getStuck$ occurs and $robotState$ changes to $Stuck$. Then, we handle the failure, for example by suspending the client for some time. For a more extensive discussion of failure handling, see [7].

Notice however, that exogenous actions like $reachDest$ are not part of the program and so the original IndiGolog interpreter cannot find a plan in such cases (there is nothing in the program that can make $robotState \neq Moving$ true). Our approach to handle this problem is that planning can be performed with a *simulated environment, modeled as an IndiGolog program*, that provides the required feedback/interaction. The environment simulator program will contain simulated exogenous actions that represent the dynamics of the environment. For the example discussed above, the following environment simulator program can be used:

proc $envSimulator$
  $< robotState = Moving \rightarrow \mathbf{sim}(reachDest)\ >$
endProc

The procedure contains an interrupt that is triggered when the robot is moving, i.e., the condition $robotState = Moving$ is true. Then, it performs $\mathbf{sim}(reachDest)$, the simulated version of the exogenous action $reachDest$. The program is run by executing $\mathbf{search}(minimizeDistance(0) \parallel envSimulator)$. That is, we run the program to accomplish the task, $minimizeDistance(0)$ (serving all clients while minimizing the distance traveled), concurrently with the environment simulator program, and search to find an execution. When an action by the environment is required during the search, the simulator program will produce it, and a plan will be found.

In the reasoning that occurs during planning, simulated actions are treated just like the corresponding real action; for any fluent F, $F(\vec{x}, do(\mathbf{sim}(a), s))$ holds if and only if $F(\vec{x}, do(a, s))$ holds, and similarly for $Poss$. But simulated actions are never executed; when the interpreter reaches a simulated action in the execution of a plan, it waits for a real exogenous action to occur in the environment.[4] If this exogenous action is the one expected, it replaces the simulated action and execution continues, otherwise replanning is performed, for example, to obtain a new plan where the unreachable client is suspended and the remaining clients are served.

More complex environment simulation programs than the one above can be used, for example one with a cooperating robot. But, our interpreter will not produce plans that work for all possible responses of a nondeterministic environment. This would require generating conditional plans. We leave this for future work.

5 Combining Reactive and Planned Behaviors

Generally, a robot working in a dynamic environment must not only perform actions to accomplish its main task, but also watch for relevant changes in the environment and react immediately to deal with these. Thus, its control program will contain a thread for accomplishing the main task that will often involve search/planning, as well as other threads to monitor for environmental changes and perform an appropriate reaction. For example, we could add to the robot control program discussed so far a separate thread to detect when a new client order has been made and send an acknowledgement to the client when that happens. The thread would run the following interrupt:

$$< o : NewOrder(o) \rightarrow ackOrder(o)\ >$$

If we run this thread at higher priority than the main thread, then when an exogenous action signaling the arrival of a new order o occurs, the robot will suspend its execution of the main plan and immediately perform the acknowledgement action for order o. Afterward, it will try to resume execution of the main plan and continue making deliveries.

Unfortunately, the original IndiGolog interpreter throws the existing plan away and searches for a new plan in such cases. Moreover, if the search/planning routine is only responsible for finding a plan for the main thread, it does not have the information required to deal with the actions that have been performed by the other reactive threads, and replanning will fail.

One way to solve this problem is to put all the threads into the search block and ask the interpreter to find a large plan that contains the actions required to accomplish the main task as well as the actions to handle exogenous actions:

proc $control$
  $\mathbf{search}(< o : NewOrder(o) \rightarrow ackOrder(o)\ >$
    $\rangle\!\rangle$ % prioritized concurrency
    $minimizeDistance(0)$
  $)$
endProc

However, searching a larger program is less efficient. Moreover, the interpreter must generate a whole new plan before it can execute the actions that react to the changes in the environment (e.g. $ackOrder$), leading to long reaction times.

We modified the IndiGolog semantics and interpreter so that it allows programs to combine a main control thread that does search and separate threads for reacting to various conditions. Then, the procedure described above can be rewritten as:

proc $control$
  $< o : NewOrder(o) \rightarrow ackOrder(o)\ >$
  $\rangle\!\rangle$
  $\mathbf{search}(minimizeDistance(0))$
endProc

[4] One can arrange for timeout exogenous actions to be generated if necessary.

This works by having a search block keep track of which primitive actions it has performed. These can then be distinguished from primitive actions performed by other reactive threads or exogenous actions. When replanning occurs, only these need to be matched to actions performed by the program in the search block. The semantics is a refinement of that presented in section 3:

$$Trans(\mathbf{search}(\delta), s, \delta', s') \equiv Trans(\mathbf{search}'(\delta, \delta, s, \emptyset), s, \delta', s')$$

$$\begin{array}{l} Trans(\mathbf{search}'(\delta, \delta_i, s_i, I), s, \delta', s') \equiv \\ \quad (\exists \gamma, \gamma').\delta' = \mathbf{search}'(\gamma', \delta_i, s_i, I') \wedge \\ \qquad Trans^*([\delta_i \parallel \delta_o(I)], s_i, [\gamma \parallel \delta_o(I)], s) \wedge \\ \qquad Trans(\gamma, s, \gamma', s') \wedge \\ \qquad (s' = s \supset I' = I) \wedge ((\exists a) s' = do(a, s) \supset I' = I \cup s') \wedge \\ \qquad (\exists \gamma'', s'').Trans^*(\gamma', s', \gamma'', s'') \wedge Final(\gamma'', s''), \end{array}$$

$$Final(\mathbf{search}(\delta), s,) \equiv Final(\mathbf{search}'(\delta, \delta, s, \emptyset), s),$$

$$\begin{array}{l} Final(\mathbf{search}'(\delta, \delta_i, s_i, I), s,) \equiv \\ \quad (\exists \gamma).Trans^*([\delta_i \parallel \delta_o(I)], s_i, [\gamma \parallel \delta_o(I)], s) \wedge Final(\gamma, s). \end{array}$$

where $\delta_o(I) \stackrel{\text{def}}{=} ((\pi a)[\textbf{if } do(a, now) \notin I \textbf{ then } a \textbf{ else } False? \textbf{ endIf}])^*$

Here, the I parameter keeps track of the actions that have occurred and came from the program *inside the search block*; its value is a set of situations $do(a, s)$ where the last action a comes from inside the search block. When we do a transition that performs an action, the resulting situation is added to I. In the path that leads from the initial situation s_i to the current situation s, we run the initial search block program δ_i concurrently with a program $\delta_o(I)$ that generates the actions that have occurred and are exogenous or came from threads outside the search block.

6 Implementation

Let us now describe how the IndiGolog interpreter of [4] has been modified to support our enhanced search mechanism. This is localized in the Prolog clauses that that implement $Trans$ and $Final$ for the search construct. The implementation differs from the semantics in a few ways. First note that instead of situations, it uses histories, which are essentially lists of actions since the initial situation. So for example, the history corresponding to situation $do(A3, do(A2, do(A1, S_0)))$ would be the list `[a3,a2,a1]`. In our implementation of the search mechanism, for efficiency, when an execution path for a search block is found, it is cached so that no additional search is required unless the path becomes invalid. This can be seen in the first clause below:

```
trans(search(E),H,E1,H1) :- findpath(E,H,P),
   trans(followpath(P,E,H,[]),H,E1,H1).

findpath(E,H,[E,H]) :- final(E,H).
findpath(E,H,[E,H|P]) :-
   trans(E,H,E1,H1), findpath(E1,H1,P).
```

The execution path found by `findpath` is saved by transforming a program `search(E)` into the construct `followpath(P,E,H,[])`, where `P` is the path found, `E` is the initial search block program, `H` is the initial history, and the last argument is a list of histories where an action from inside the search block has just been performed, initially `[]`.

Then, we follow the cached path without rechecking it as long as the current history matches that present in the cached path:

```
trans(followpath([_E,H,E1,H|P],EI,HI,I),H,
      followpath([E1,H|P],EI,HI,I),H) :- !.
      /* no action done */
trans(followpath([_E,H,E1,[A|H]|P],EI,HI,I),H,
      followpath([E1,[A|H]|P],EI,HI,[[A|H]|I]),
      [A|H]) :- !. /* does action A */
```

If an outside action (or several) occurs and the current history `CH` differs from that in the cached path `H`, then we first check if the cached path still works (i.e., still leads to a final situation); if so, we fix it so that the histories in it include the outside action(s), and continue following it:

```
trans(followpath([E,H,E2,H2|P],EI,HI,I),
      CH,E1,H1) :-
   canfixpath(CH,[E,H,E2,H2|P],P1),
   trans(followpath(P1,EI,HI,I),CH,E1,H1), !.

canfixpath(CH,[E,_H],[E,CH]) :- final(E,CH).
canfixpath(CH,[E,H,E1,H1|P],[E,CH,E1,CH1|P1]):-
   trans(E,CH,E1,CH1),
   canfixpath(CH1,[E1,H1|P],[E1,CH1|P1]).
```

Otherwise, we must redo the search from the original program and history. The code for this is as follows:

```
trans(followpath([_E,_H,_E1,_H1|_P],EI,HI,I),
      CH,E1,H1) :-
   append(I,[HI],I1),
   extractAct([CH|I1],[],AL),
   transStarCompat(AL,EI,HI,E2,CH),
   findpath(E2,CH,P),
   trans(followpath(P,EI,HI,I),CH,E1,H1).

extractAct([H,H|L],AL,R) :- !,
   extractActIn([H|L],AL,R).
extractAct([[A|H1],H|L],AL,R) :-
   extractAct([H1,H|L],[A|AL],R).

extractActIn([H0],AL,AL).
extractActIn([[A|H1],H|L],AL,R) :-
   extractAct([H1,H|L],[inside(A)|AL],R).

transStarCompat([],E,H,E,H).
transStarCompat([],E,H,E1,H) :-
   trans(E,H,E2,H),
   transStarCompat([],E2,H,E1,H).
transStarCompat([inside(A)|L],E,H,E1,H1) :-
   prim_action(A), trans(E,H,E2,[A|H]),
   transStarCompat(L,E2,[A|H],E1,H1).
transStarCompat([A|L],E,H,E1,H1) :-
   A \= inside(_A),
   transStarCompat(L,E,[A|H],E1,H1).
transStarCompat([A|L],E,H,E1,H1) :-
   trans(E,H,E2,H),
   transStarCompat([A|L],E2,H,E1,H1).
```

In redoing the search from the original program `EI` and original history `HI`, we only want to consider paths that are consistent with the actions that have occurred so far. To do this, we first make a list `AL` of the actions that have occurred so far using `extractAct` (and `extractActIn`), labeling the ones that have come from inside the search block with `inside(Action)`. Then, we use `transStarCompat` to find a sequence of transition compatible

with the actions in AL from the original program EI and original history HI to a new remaining program E2 and the current history CH. After that, we try to find a terminating execution path P for the new remaining program E2, and if we do, we cache it and return the first transition from it.

The clauses implementing $Final$ appear below, and use no ideas beyond those introduced above:

```
final(search(E),H) :- final(E,H).

final(followpath([_,H],_EI,_HI,_I),H) :- !.
/* off path; first check if final */
final(followpath([E,_H],_EI,_HI,_I),CH) :-
   final(E,CH), !.
final(followpath([E,_H|_P],_EI,_HI,_I),CH) :-
   final(E,CH), !.
/* else check if current history is final */
/* for some execution of original prog */
final(followpath(_P,EI,HI,I),CH):-
   append(I,[HI],I1),
   extractAct([CH|I1],[],AL),
   transStarCompat(AL,EI,HI,E2,CH),
   final(E2,CH).
```

7 Conclusion and Future Work

In this paper, we have shown that an enhanced version of the IndiGolog programming language supports the development of effective high-level control programs for robots that operate in dynamic and incompletely known environments. Such programs effectively integrate planning, sensing the environment, and reactive plan execution to cope with relevant environmental changes.

The main enhancements that were made to IndiGolog are the following:

- In replanning, the search starts from the initial program and situation instead of the advanced program and situation. As a result, there is no commitment to a particular branch in the user program. Solutions can now be found for cases such as our iterative deepening route planner where the original interpreter would fail.
- A simulated environment facility was added to support planning in a context where the agent must rely on the occurrence of exogenous actions in the environment.
- Programs that combine threads that do search/planning with separate threads that react to events can now be handled. This is done by having search blocks keep track of the actions that are performed by the block, and using this information in replanning. One can then build robot controllers that react quickly and still do planning.

We have already tested our enhanced IndiGolog interpreter and versions of the example robot control programs described earlier with a simulated operating environment. We will soon be testing them with a real Nomad Super Scout II robot operating in a real environment. As in [7], the system will consist of a low-level reactive module that performs path following and collision avoidance, a path planning module, and a high-level deliberative module involving the IndiGolog interpreter running the high-level robot control program. These modules will be running asynchronously to control the robot.

In future work, we would like to extend our current planning with a simulated environment facility to support true contingent planning [14, 5], i.e., the synthesis of plans that include conditional branching (in the style of [8]) to deal with sensing results and unpredictable environments. We would also like to support the generation of plans that contain sensing actions. We are also exploring the use of a reflective interpreter as a way of providing the programmer with greater control over the IndiGolog planning and execution facility. As well, we are working on a multi-robot version of the mail delivery application, where each robot has its own IndiGolog controller, and the robots cooperate to accomplish the task.

Acknowledgements

This research received financial support from Communications and Information Technology Ontario and the Natural Science and Engineering Research Council of Canada. We thank Hector Levesque and Michael Jenkin for their comments. Hector provided help with the Prolog implementation.

REFERENCES

[1] W. Burgard, A.B. Cremers, D. Fox, D. Haehnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun, 'The interactive museum tour-guide robot', in *Proceedings of the 15th National Conference on Artificial Intelligence*. AAAI Press, (July 1998).

[2] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque, 'Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus', in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 1221–1226, Nagoya, Japan, (August 1997).

[3] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque, 'ConGolog, a concurrent programming language based on the situation calculus', *Artificial Intelligence*, (2000). To Appear.

[4] Giuseppe De Giacomo and Hector J. Levesque, 'An incremental interpreter for high-level programs with sensing', in *Logical Foundations for Cognitive Agents*, eds., Hector J. Levesque and Fiora Pirri, 86–102, Springer-Verlag, Berlin, Germany, (1999).

[5] E. Guere and R. Alami, 'A possibilistic planner that deals with non-determinism and contingency', in *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pp. 996–1001, Stockholm, (August 1999).

[6] M. Hennessy, *The Semantics of Programming Languages*, John Wiley & Sons, 1990.

[7] Yves Lespérance, Kenneth Tam, and Michael Jenkin, 'Reactivity in a logic-based robot programming framework', in *Intelligent Agents VI — Agent Theories, Architectures, and Languages, 6th International Workshop, ATAL'99, Proceedings*, eds., N. Jennings and Y. Lespérance, volume 1757 of *LNAI*, 173–187, Springer-Verlag, (2000).

[8] Hector J. Levesque, 'What is planning in the presence of sensing?', in *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 1139–1146, Portland, OR, (August 1996).

[9] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl, 'GOLOG: A logic programming language for dynamic domains', *Journal of Logic Programming*, **31**(59–84), (1997).

[10] John McCarthy and Patrick Hayes, 'Some philosophical problems from the standpoint of artificial intelligence', in *Machine Intelligence*, eds., B. Meltzer and D. Michie, volume 4, 463–502, Edinburgh University Press, Edinburgh, UK, (1979).

[11] G. Plotkin, 'A structural approach to operational semantics', Technical Report DAIMI-FN-19, Computer Science Dept., Aarhus University, Denmark, (1981).

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

[13] K. Tam, J. Lloyd, Y. Lespérance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and M. Jenkin, 'Controlling autonomous robots with GOLOG', in *Proceedings of the Tenth Australian Joint Conference on Artificial Intelligence (AI-97)*, pp. 1–12, Perth, Australia, (November 1997).

[14] D.S. Weld, C.R. Anderson, and D.E. Smith, 'Extending graphplan to handle uncertainty and sensing actions', in *Proceedings of the 15th National Conference on Artificial Intelligence*, pp. 897–904. AAAI Press, (July 1998).