Incremental Plan Recognition in an Agent Programming Framework

Alexandra Goultiaeva
Department of Computer Science
University Of Toronto
Toronto, ON Canada M5S 1A4
alexia@cs.toronto.edu

Yves Lespérance
Dept. of Computer Science and Engineering
York University
Toronto, ON Canada M3J 1P3
lesperan@cse.yorku.ca

Abstract

In this paper, we propose a formal model of plan recognition for inclusion in a cognitive agent programming framework. The model is based on the Situation Calculus and the Con-Golog agent programming language. This provides a very rich plan specification language. Our account also supports incremental recognition, where the set of matching plans is progressively filtered as more actions are observed. This is specified using a transition system account. The model also supports hierarchically structured plans and recognizes sub-plan relationships.

Introduction

The ability to recognize plans of others can be useful in a wide variety of applications, from office assistance (where a program might provide useful reminders, or give hints on how to correct a faulty plan), to monitoring and aiding astronauts, providing assistance to people with cognitive or memory problems to allow them to live independently, etc.

There has been a lot of work in the area of plan recognition; see (Carberry 2001) for a recent survey. Some of this work develops symbolic techniques for identifying plans that match the observations. For instance, (Avrahami-Zilberbrand & Kaminka 2005) uses a decision tree to match observations to plan steps and graph traversal to identify branches that represent consistent hypotheses. To deal with uncertainty and identify most likely hypotheses, some work uses probabilistic techniques; for instance (Bui 2003), uses an extension of Hidden Markov Models for this. Other work combines symbolic and probabilistic approaches, e.g. (Avrahami-Zilberbrand & Kaminka 2006). Many approaches (including the ones just cited) support hierarchical task network-type plans, allowing methods to have several alternative decompositions, as well as looping tasks. However, these approaches do not support concurrently executing plans.

Our approach is based on the ConGolog agent programming language (De Giacomo, Lespérance, & Levesque 2000), which supports very rich plans, including concurrent processes. We think that developing a unified agent programming framework that supports plan recognition as well as plan synthesis and behavior specification would have a number of benefits, including ease of use, and reuse of domain specifications and reasoning methods. Our formalization of plan recognition also handles cases where there is only incomplete knowledge about the initial situation. Probably the closest related work is (Demolombe & Hamon 2002), where plans are represented as Golog[1] programs, with two additional constructs: σ, which matches any sequence of actions, and $\alpha_1 - \alpha_2$, which matches an execution of plan α_1 as long as it does not also match an execution of α_2. $\alpha_1 - \alpha_2$ is quite a useful and powerful construct, which allows one to specify plans in terms of what must not happen in addition to what can happen. In some sense, the programs that we put in the plan library can be "plan recognition templates", rather than actual plans that an agent would use. This cannot be done in most other plan recognition frameworks. It can also be useful for monitoring applications, where one wants to represent policies and detect violations.

In this paper, we provide an alternative formalization and implementation of the plan recognition framework of (Demolombe & Hamon 2002). Plans are represented as procedures, which may include calls to other procedures. Because of this, the plan recognition framework provides additional information, such as the call hierarchy, which details the procedures that are in progress or have completed, which procedure called which, and what remains to execute.

Another major difference between our approach and that of (Demolombe & Hamon 2002) is that we support *incremental plan recognition*. Given a set of hypotheses about what plans may be executing and a new observed action, our formalization defines what the revised set of hypotheses should be. Plan recognition is specified in terms of a structural operational semantics (single-step transitions) in the style of (Plotkin 1981) for the plan specification language. (Demolombe & Hamon 2002) used a different semantics where programs were mapped into complete executions.

We have implemented a plan recognition system based on this formalization. It can be executed "on-line" and constantly keeps track of what plans may be executing, without having to recalculate them for each new observed action. Focusing on procedures rather than complete plans allows

---

[1]Golog (Levesque *et al.* 1997a) is a precursor of ConGolog that does not support concurrency.

plans to be hierarchical and modular, and the result of the recognition is more informative and meaningful.

In the rest of the paper, we first give an overview of the Situation Calculus and ConGolog, and then present our formal model of plan recognition. Then, we give some examples to illustrate how the framework is used. Following this, we briefly describe our implementation of the model. We conclude the paper with a discussion of the novel features and limitations of our account, and provide suggestions for future work.

The Situation Calculus and ConGolog

The technical machinery that we use to define high-level program execution is based on that of (De Giacomo, Lespérance, & Levesque 2000). The starting point in the definition is the situation calculus (McCarthy & Hayes 1979). We will not go over the language here except to note the following components: there is a special constant S_0 used to denote the *initial situation*; there is a distinguished binary function symbol do where $do(a, s)$ denotes the successor situation to s resulting from performing the action a; relations whose truth values vary from situation to situation, are called (relational) *fluents*, and are denoted by predicate symbols taking a situation term as their last argument. There is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s.

Within this language, we can formulate domain theories which describe how the world changes as a result of the available actions. Here, we use action theories of the following form:

- Axioms describing the initial situation, S_0.

- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.

- 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 (Reiter 1991).

- Unique names axioms for the primitive actions.

- A set of foundational, domain independent axioms for situations Σ as in (Reiter 2001).

Next we turn to programs. The programs we consider here are based on the ConGolog language defined in (De Giacomo, Lespérance, & Levesque 2000), providing a rich set of programming constructs as follows:

α,	primitive action
$\phi?$,	wait for a condition
$\delta_1; \delta_2$,	sequence
$\delta_1 \mid \delta_2$,	nondeterministic branch
$\pi\, x.\, \delta$,	nondeterministic choice of argument
δ^*,	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**,	conditional
while ϕ **do** δ **endWhile**,	while loop
$\delta_1 \parallel \delta_2$,	concurrency with equal priority
$\delta_1 \rangle\!\rangle\, \delta_2$,	concurrency with δ_1 at a higher priority
$\delta^\parallel$,	concurrent iteration
$\langle\, \phi \rightarrow \delta\,\rangle$,	interrupt
$p(\vec{\theta})$,	procedure call

Among these constructs, we notice the presence of nondeterministic constructs. These include $(\delta_1 \mid \delta_2)$, which nondeterministically chooses between programs δ_1 and δ_2, $\pi\, x.\, \delta$, which nondeterministically picks a binding for the variable x and performs the program δ for this binding of x, and δ^*, which performs δ zero or more times. Also notice that ConGolog includes constructs for dealing with concurrency. In particular $(\delta_1 \parallel \delta_2)$ denotes the concurrent execution (interpreted as interleaving) of the programs δ_1 and δ_2.

In (De Giacomo, Lespérance, & Levesque 2000), a single step transition semantics in the style of (Plotkin 1981) is defined for ConGolog programs. Two special predicates $Trans$ and $Final$ are introduced. $Trans(\delta, s, \delta', s')$ means that by executing program δ starting in situation s, one can get to situation s' in one elementary step with the program δ' remaining to be executed. $Final(\delta, s)$ means that program δ may successfully terminate in situation s.

Offline executions of programs, which are the kind of executions originally proposed for Golog and ConGolog (Levesque *et al.* 1997b; De Giacomo, Lespérance, & Levesque 2000), are characterized using the $Do(\delta, s, s')$ predicate, which means that there is an execution of program δ (a sequence of transitions) that starts in situation s and terminates in situation s':

$$Do(\delta, s, s') \overset{\text{def}}{=} \exists \delta'.Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s'),$$

where $Trans^*$ is the reflexive transitive closure of $Trans$. An offline execution of δ from s is a sequence of actions $a, \ldots, a_n$ such that: $\mathcal{D} \cup \mathcal{C} \models Do(\delta, s, do(a_n, \ldots, do(a_1, s) \ldots))$, where $\mathcal{D}$ is an action theory as mentioned above, and $\mathcal{C}$ is a set of axioms defining the predicates $Trans$ and $Final$ and the encoding of programs as first-order terms (De Giacomo, Lespérance, & Levesque 2000).

Formalizing plan recognition

Recognizing a plan means that given a sequence of observed actions, the system must be able to determine which plan(s) the user may be following. The framework described here relies on a plan library, which details the possible plans as procedures in ConGolog. Given the sequence of actions performed, the system should be able to provide the following information: the plan that the user is currently following; the stage in the plan that the user is following – what has already been done and what remains to be done; and which procedures that plan is part of – is the user doing it as part of a larger plan?

The framework is specified in terms of ConGolog, to which a few extensions are made. Note that what is described below could have alternatively been done by modifying the semantics of the language. The following formalization is designed to build on top of the existing framework as much as possible.

First, we introduce two special primitive actions: $startProc(name(args))$ and $endProc(name(args))$. These are *annotation actions*, present only in the plan library, but never actually observed. The two actions are used to represent procedure invocation and completion. It is assumed that every procedure that we want to distinguish in the plan library starts with the

action $startProc(name(args))$ and ends with the action $endProc(name(args))$, where $name$ is the name of the procedure in which the actions occur, and $args$ are its arguments. This markup can be generated automatically given a plan library.

Our transition system semantics for plans fully supports concurrency. Environments involving multiple agents can also be dealt with if we assume that the agent of each action is specified (say as a distinguished parameter of the action). However, if there is concurrency over different procedures run by the same agent, the annotated situation as currently defined is not generally sufficient to determine which thread/procedure an observed action belongs to. Additional annotations will need to be introduced to specify this. We leave this for future work.

After the inclusion of the annotation actions, for each sequence of actions there are two situations: the real (observed) situation, and the annotated situation, which includes the actions $startProc$ and $endProc$. Given the annotated situation, it is straightforward to obtain the state of the execution stack (which procedures are currently executing), determine what actions were executed by which procedures, and determine the remaining plan. An action $startProc(proc)$ means that the procedure $proc$ was called, and should be added to the stack. The action $endProc(proc)$ signals that the last procedure has terminated, and should be removed from the stack. Note that for a given real situation, there may be multiple annotated situations that would match it. Each of those situations would show a different possible execution path in the plan library. For example, if the plan library contained the following procedures:

proc $p1$ $startProc(p1); a; b; endProc(p1)$ **endProc**
proc $p2$ $startProc(p2); a; c; endProc(p2)$ **endProc**

then the real situation $do(a, S_0)$ would have two possible annotated situations that would match it: $do(a, do(startProc(p1), S_0))$ and $do(a, do(startProc(p2), S_0))$. In this context, the plan recognition problem reduces to the following: given the observed situation and a plan library, find the possible annotated situations.

The first two predicates defined for the new formalism are $aTrans$ and $rTrans$. The predicate $aTrans$ is a form of $Trans$ that allows only a transition step that cannot be observed: either an annotation action or a test/wait action. The predicate $rTrans$ is a form of $Trans$ which only allows observable actions. The helper predicate $Annt$ is true if and only if the action passed to it is an annotation action:

$$Annt(a) \stackrel{\text{def}}{=} \exists n \,.\, a = startProc(n) \vee \exists n \,.\, a = endProc(n)$$
$$aTrans(\delta, s, \delta', s') \stackrel{\text{def}}{=}$$
$$\quad Trans(\delta, s, \delta', s') \wedge$$
$$\quad (\exists a \,.\, (s' = do(a, s) \wedge Annt(a)) \vee s' = s)$$
$$rTrans(\delta, s, \delta', s') \stackrel{\text{def}}{=}$$
$$\quad Trans(\delta, s, \delta', s') \wedge \exists a \,.\, s' = do(a, s) \wedge \neg Annt(a)$$

We also define $aTrans^*$ as the reflexive transitive closure of $aTrans$.

The transition predicate $nTrans(\delta, s_r, s_a, \delta', s_r', s_a')$ is the main predicate in our plan recognition framework. It

holds when δ' is the program remaining from δ after performing any number of annotation actions or tests, followed by an observable action. Situation s_r is the real situation before performing those steps, and s_r' is the real situation after. Situation s_a is the annotated situation (which reflects the annotations as well as the real actions) before the program steps, and s_a' is the annotated situation after. Effectively, our definition below amounts to $nTrans$ being equivalent to $aTrans^*$ composed with $rTrans$:

$$nTrans(\delta, s_r, s_a, \delta', s_r', s_a') \stackrel{\text{def}}{=}$$
$$(\exists \delta'', s_a'', a).aTrans^*(\delta, s_a, \delta'', s_a'')$$
$$\wedge rTrans(\delta'', s_r, \delta', do(a, s_r))$$
$$\wedge s_r' = do(a, s_r) \wedge s_a' = do(a, s_a'').$$

Just as $nTrans$ is the counterpart to $Trans$ which deals with annotation actions, $nFinal$ is the counterpart to $Final$, which allows any number of annotation actions or tests to be performed:

$$nFinal(\delta, s) \stackrel{\text{def}}{=}$$
$$(\exists \delta', s')(aTrans^*(\delta, s, \delta', s') \wedge Final(\delta', s'))$$

As mentioned in (Demolombe & Hamon 2002), in many cases it would be useful for the procedures to leave some actions unspecified, or to place additional constraints on the plans. So they introduced two new constructs. The first is $anyBut(actionList)$, which allows one to execute an arbitrary primitive action which is not in its argument list. For example, $anyBut([b, d])$ would match actions a or c, but not b or d. It is a useful shorthand for writing general plans which might involve unspecified steps. For example, a plan might specify that a certain condition needs to hold for its continuation, but leave unspecified what action(s) was performed to achieve the condition. It is simply an abbreviation, included for convenience. Another shorthand construct, any, can be defined to match any action without exceptions. We can define these as follows:[2]

$$anyBut([a_1, ..., a_n]) \stackrel{\text{def}}{=}$$
$$\quad \pi a.(\textbf{if}(a \neq a_1 \wedge ... \wedge a \neq a_n) \textbf{ then } a \textbf{ else } false?\textbf{endIf})$$
$$any \stackrel{\text{def}}{=} anyBut([])$$

The second construct is $minus(\delta, \hat{\delta})$. This matches any execution that would match δ, as long as it does not match $\hat{\delta}$. This construct allows the plan to place additional constraints on the sequences of actions that would be recognized within a certain procedure. For example, the procedure that corresponds to a task of cleaning the house could include unspecified parts, and would match many different sequences of actions, but not if they involve brushing teeth. Assuming $cleanUp$ and $brushTeeth$ are procedures in the plan library, then it is possible to specify the above as $minus(cleanUp, brushTeeth)$.

To define this construct, we need to define what a step of execution for this construct is, and the remaining program. Also, note that $\hat{\delta}$ must match all observable actions performed by δ, but might do different annotation and test actions; those differences should be ignored.

---

[2]When $n = 0$, by convention the condition is equivalent to $True$.

An additional axiom is added to specify $Trans$ for the $minus$ construct:

$$Trans(minus(\delta, \hat{\delta}), s, \delta', s') \equiv$$
$$\exists \delta''.aTrans(\delta, s, \delta'', s') \wedge \delta' = minus(\delta'', \hat{\delta})$$
$$\vee \ (\exists \delta''a.rTrans(\delta, s, \delta'', do(a, s)) \wedge s' = do(a, s)$$
$$\wedge \ (\neg \exists \hat{\delta}' s'' s_i.nTrans'(\hat{\delta}, s, s, \hat{\delta}', do(a, s''), s_i) \wedge \delta' = \delta''$$
$$\vee \ \exists \hat{\delta}' s'' s_i.nTrans'(\hat{\delta}, s, s, \hat{\delta}', do(a, s''), s_i)$$
$$\wedge \neg nFinal'(\hat{\delta}', do(a, s'')) \wedge \delta' = minus(\delta'', \hat{\delta}')).$$

This says the following: if the next step of the plan δ is not an observable action, then the remaining program is what remains of δ minus $\hat{\delta}$; if δ performs an observable action, and $\hat{\delta}$ cannot match that action, then the remaining program is what remains of δ; if $\hat{\delta}$ can match the observable action performed by δ but it is not final, then the remaining program is what remains of δ minus what remains of $\hat{\delta}$.

Note that whether $Trans$ holds for $minus(\delta, \hat{\delta})$ depends on whether $nTrans$ holds for $\hat{\delta}$ and the latter depends on $aTrans^*$ and ultimately $Trans$, so the definition might not appear to be well founded. We ensure that it is well founded by imposing the restriction that no $minus$ can appear in the second argument $\hat{\delta}$ of a $minus$. So in the axiom, we use $nTrans'$ which is defined just like $nTrans$, except that it is based on a version of $Trans$, $Trans'$, that does not support the $minus$ construct and does not include the $Trans$ axiom for the $minus$ construct. So $Trans'$ is just the existing $Trans$ from (De Giacomo, Lespérance, & Levesque 2000), which is well defined, and $nTrans'$ is defined in terms of it. Then we can define the new $Trans$ that supports $minus$ in terms of $nTrans'$ and we have a well founded definition. The same approach is used to define $Final$ for $minus$. The construct $minus$ is considered finished when δ is finished, but $\hat{\delta}$ is not:

$$Final(minus(\delta, \hat{\delta}), s) \equiv Final(\delta, s) \wedge \neg nFinal'(\hat{\delta}, s).$$

We use $\mathcal{C}'$ to denote the extended ConGolog axioms: $\mathcal{C}$ together with the above two. Note that recursive procedures can be handled as in (De Giacomo, Lespérance, & Levesque 2000).

The above definition relies on a condition imposed on the $\hat{\delta}$ that may appear as second argument in a $minus$: for any sequence of transitions involving the same actions, $\hat{\delta}$ should have only one possible remaining program. More formally:

$$Trans^*(\hat{\delta}, s, \hat{\delta}_1, s_1) \wedge Trans(\hat{\delta}_1, s_1, \hat{\delta}', do(a_1, s_1)) \wedge$$
$$Trans^*(\hat{\delta}, s, \hat{\delta}_2, s_2) \wedge Trans(\hat{\delta}_2, s_2, \hat{\delta}'', do(a_2, s_2)) \wedge$$
$$do(a_1, s_1) = do(a_2, s_2)$$
$$\supset \hat{\delta}' = \hat{\delta}''$$

This restriction seems quite natural because $\hat{\delta}$ is a model of what is not allowed. If there are many possibilities about what is not allowed after a given sequence of transitions, then the model seems ill formed or at least hard to work with. An example of what is not allowed as $\hat{\delta}$ would be the program $(a; b)|(a; c)$, because after observing the action a, there could be two possible remaining programs: b or c. Then we have

$$Trans(minus((a; c), (a; b)|(a; c)), s, minus(c, b), do(a, s))$$

which is wrong because $a; c$ is also ruled out. If rewritten as $a; (b|c)$, this program is allowed. [3]

Based on the above definition, to get the annotated situation from an observable one, we only need to apply $nTrans$ a number of times, until the observable situation is reached. We define $nTrans^*$ as the reflexive transitive closure of $nTrans$. The predicate $allTrans(s_r, s_a, \delta_{rem})$ means that s_a denotes a possible annotated situation that matches the observed situation s_r, and δ_{rem} is the remaining plan:

$$allTrans(s_r, s_a, \delta_{rem}) \overset{\text{def}}{=}$$
$$nTrans^*(planLibrary, S_0, S_0, \delta_{rem}, s_r, s_a),$$

where S_0 is the initial situation and $planLibrary$ is a procedure that represents the plan library.

The set of all the remaining programs δ and their corresponding annotated situations S_a that are consistent with a given observed situation S and domain action theory $\mathcal{D}$ can be defined as follows:

$$allConsistPlans(S) \overset{\text{def}}{=} \{(\delta, S_a)|$$
$$\mathcal{D} \cup \mathcal{C}' \cup \{allTrans(S, S_a, \delta)\} \text{ is consistent}\}.$$

Our transition system semantics defines program executions incrementally. So we have that for any model M, $M \models \mathcal{D} \cup \mathcal{C}' \cup \{allTrans(do(A, S), S_a, \delta)\}$ if and only if $M \models \mathcal{D} \cup \mathcal{C}' \cup \{\exists \delta', s_a'[allTrans(S, s_a', \delta') \wedge nTrans(\delta', S, s_a', \delta, do(A, S), S_a)]\}$. This can be used to incrementally calculate the set of plans that the agent may be executing under some conditions. Suppose that for every hypothesis $(\delta, S_a) \in allConsistPlans(S)$, we can compute a condition $\gamma_{S_a, \delta}$ on the initial situation such that $\mathcal{D} \cup \mathcal{C}' \models allTrans(S, S_a, \delta) \equiv \gamma_{S_a, \delta}(S_0)$. Actually, $\gamma_{S_a, \delta}$ can be obtained in a straightforward way from $allTrans(S, S_a, \delta)$ by using the ConGolog axioms to eliminate the ConGolog operators and then computing the regression of the result. Then, we have that[4]

$$allConsistPlans(do(A, S)) = \{(\delta', S_a')|$$
$$(\delta, S_a) \in allConsistPlans(S) \text{ and } \mathcal{D} \cup \mathcal{C}' \cup$$
$$\{\gamma_{S_a, \delta}(S_0) \wedge nTrans(\delta, S, S_a, \delta', do(A, S), S_a')\}$$
$$\text{is consistent }\}.$$

Note that if there is complete information about the initial situation, then computing hypotheses incrementally is even simpler. In this case, there is a unique model M_0 such that $M_0 \models \mathcal{D}$. Then we have that

$$allConsistPlans(do(A, S)) = \{(\delta', S_a')|$$
$$(\delta, S_a) \in allConsistPlans(S) \text{ and}$$
$$M_0 \models nTrans(\delta, S, S_a, \delta', do(A, S), S_a')\}.$$

---

[3]We could try to drop this restriction and collect all the remaining $\hat{\delta}$, but it is not clear that these can always be finitely represented, e.g. $\pi n.(PositiveInteger(n)?; a; b(n))$.

[4]This also requires a domain closure assumption: if $\mathcal{D} \cup \mathcal{C}' \cup \{\exists \delta, s_a \ allTrans(S, s_a, \delta)\}$ is consistent, then there exist ground terms δ and S_a such that $\mathcal{D} \cup \mathcal{C}' \cup \{allTrans(S, S_a, \delta)\}$ is consistent.

Examples

The main example described here is a simulation of activities in a home. There are four rooms: the bedroom, kitchen, living room, and bathroom. There are also four objects: the toothbrush, book, spoon, and cup. Each object has its own place, where it should be located. The toothbrush should be in the bathroom, the book in the living room, and the spoon and cup in the kitchen.

Initially, all objects are where they are supposed to be, except for two: the book is in the kitchen, and the toothbrush is in the living room. The location of the monitored agent is originally in the bedroom. Note that in this case, we have complete knowledge about the initial situation. This is required by our current implementation.

There are four possible primitive actions:

- $goTo(room)$: changes the location of the agent to be $room$;
- $pickUp(object)$: only possible if the agent is in the same room as the object; this causes the object to be held;
- $putDown(object)$: only possible if the agent holds the object; puts the object down;
- $use(object)$: only possible if the agent holds the object.

We use the following fluents:

- loc: the room in which the agent is;
- $loc(thing)$: the room in which the thing is;
- $Hold(thing)$: true if the agent holds the thing, false otherwise.

We also use the following non-fluent predicates:

- $Room(r)$: r is a room;
- $Object(t)$: t is an object;
- $InPlace(thing, room)$: holds if $thing$ is in its place when it is in $room$.

There are five procedures in the plan library:

- $get(thing)$: go to the room where thing is, and pick it up;
- $putAway(thing)$: go to the room where the thing should be, and put it down;
- $cleanUp$: while there are objects that are not in their places, get such an object, or put it away;
- $brushTeeth$: get the toothbrush, use the toothbrush, and either put away the toothbrush, or put it down (where the agent is);
- $readBook$: get the book, use the book, and either put away the book, or put it down.

The procedures are defined below. We also use the following procedure:

proc $getTo(r)$
 $Room(r)?;$ **if** $loc \neq r$ **then** $goTo(r)$**endIf**
endProc

$getTo$ checks if the current location is already the destination room r. If not, the action $goTo$ is executed. It is a helper procedure, which was only introduced for convenience, and

was not deemed important enough to appear in the annotations. Hence, it does not have $startProc$ and $endProc$ actions. So, when the program is executed, the procedure $getTo$ will not appear in the stack.

The definition of most of the other procedures is straightforward:

proc $get(t)$
 $startProc(get(t)); (\neg Hold(t))?; getTo(loc(t));$
 $pickUp(t); endProc(get(t))$
endProc;
proc $putAway(t)$
 $startProc(putAway(t)); (Hold(t))?;$
 $\pi\, r.InPlace(t, r)?; getTo(r); putDown(t);$
 $endProc(putAway(t))$
endProc;
proc $brushTeeth$
 $startProc(brushTeeth); get(toothbrush);$
 $use(toothbrush);$
 $(putAway(toothbrush)|putDown(toothbrush));$
 $endProc(brushTeeth)$
endProc;
proc $readBook$
 $startProc(readBook); get(book); use(book),$
 $(putAway(book)|putDown(book));$
 $endProc(readBook)$
endProc;

Procedures $brushTeeth$ and $readBook$ have options: either the agent might put the thing away in its place, or it might put the thing down wherever it happens to be. In practice, a person might do either, and both executions should be recognized as part of the procedure.

Perhaps the most complex procedure in this example is $cleanUp$. The main idea is that when executing this procedure, the agent will, at each iteration, get a thing that is not in its proper place, or put away something it already holds.

proc $cleanUp$
 $startProc(cleanUp);$
  while $\exists t.Object(t) \wedge \neg InPlace(t, loc(t))$ **do**
   $\pi\, t.Object(t) \wedge \neg InPlace(t, loc(t))?;$
    $(get(t)|putAway(t))$
  endWhile;
 $endProc(cleanUp)$
endProc

The main plan library chooses some procedure to execute nondeterministically and repeats this zero or more times:

proc $planLibrary$
 $(cleanUp|(brushTeeth|(readBook|(\pi t.get(t)))))^*.$
endProc

Let's look at an execution trace for the above example. Suppose that the first action was $goTo(kitchen)$. The following possible scenarios are then output by the system:

```
proc get(book)  -> goTo(kitchen)
proc get(cup)   -> goTo(kitchen)
proc get(spoon) -> goTo(kitchen)
proc readBook   -> proc get(book)
   -> goTo(kitchen)
proc cleanUp    -> proc get(book)
   -> goTo(kitchen)
```

The system is trying to guess what the user is doing by going to the kitchen. It lists the five plans from the library that might have this first action. Note that the possibilities of doing $cleanUp$ by getting a cup or a spoon are not listed. This is because both the spoon and cup are already in their places, so if the agent picked them up, it would not be cleaning up.

Now suppose that the next action is $pickUp(book)$. Then, the system can discard some of the above possibilities, namely those which involve taking something else. The new possible scenarios are:

```
proc get(book)
   -> goTo(kitchen); pickUp(book)
proc readBook -> get(book)
   -> goTo(kitchen); pickUp(book)
proc cleanUp -> proc get(book)
   -> goTo(kitchen); pickUp(book)
```

The next action is $use(book)$. The plan $get(book)$ is finished, but there is no plan in the library that could start with the action $use(book)$. So, this possibility can be discarded. The next action of cleanUp cannot match the observed actions as well. Thus the only remaining possible plan is $readBook$:

```
proc readBook -> proc get(book)
   -> goTo(kitchen); pickUp(book);
   use(book)
```

Now, let us consider a different scenario. In order to demonstrate the use of the $minus$ and $anyBut$ constructs, we can define two variants of $cleanUp$. In the first one, $cleanUp_u$, an arbitrary action is allowed at the end of every iteration of the loop. The second one, $cleanUp_m$, together with the optional arbitrary action, introduces a constraint: a sequence of actions will not be matched if it involves the execution of procedure $brushTeeth$. This is achieved by using the $minus$ construct.

> **proc** $cleanUp_u$
> $startProc(cleanUp_u)$;
>   **while** $\exists t.Object(t) \wedge \neg InPlace(t, loc(t))$ **do**
> $\pi\, t.Object(t) \wedge \neg InPlace(t, loc(t))?;$
> $(get(t)|putAway(t)); (any|nil)$
>   **endWhile**;
> $endProc(cleanUp_u)$
> **endProc**
> **proc** $cleanUp_m$
> $startProc(cleanUp_m)$;
> $minus($
>   **while** $\exists t.Object(t) \wedge \neg InPlace(t, loc(t))$ **do**
> $\pi\, t.Object(t) \wedge \neg InPlace(t, loc(t))?;$
> $(get(t)|putAway(t)); (any|nil)$;
>   **endWhile**,
> $[brushTeeth])$;
> $endProc(cleanUp_m)$
> **endProc**

Suppose that the sequence of observed actions starts with the two actions $goTo(livingRoom)$ and $take(toothbrush)$. All three variants of $cleanUp$ would match those actions, and produce the same scenario:

```
proc cleanUp_k -> proc get(toothbrush)
   goTo(livingRoom); pickUp(toothbrush)
```

where k is either nothing, or u or m, depending on the version of the program used.

Now suppose that the next action is $use(toothbrush)$. The original version of $cleanUp$ does not match the observed action. The other two variants, $cleanUp_u$ and $cleanUp_m$, would still match the situation, because the new action matches the unspecified action at the end of the loop.

If the next action is $goTo(bathroom)$, then both remaining procedures match this as well:

```
proc cleanUp_k -> proc get(toothbrush)
   -> goTo(livingRoom);
   pickUp(toothbrush); use(toothbrush)
   proc putAway(toothbrush)
      -> goTo(bathroom)
```

where k can only be u or m.

Now, if the next step is $putDown(toothbrush)$, then $cleanUp_u$ matches it. However, $cleanUp_m$ does not. That is because $cleanUp_m$ has the minus construct, and the observed actions matched the exception part of it. The action $putDown(toothbrush)$ can be considered the last action of $brushTeeth$, which was ruled out by the $minus$ in $cleanUp_m$. So, $cleanUp_m$ cannot match this sequence of actions. $cleanUp_u$, which is identical to $cleanUp_m$ except for the $minus$ construct, does match the action, and produces the following scenario:

```
proc cleanUp_u -> proc get(toothbrush)
   -> goTo(livingRoom), pickUp(toothbrush)
   use(toothbrush)
   proc putAway(toothbrush)
   -> goTo(bathroom), putDown(toothbrush)
```

Another example that the system was tested on is that from (Demolombe & Hamon 2002) involving aircraft flying procedures. There is a single procedure called $fireOnBoard$. It involves three actions, performed sequentially, with possibly other actions interleaved. The three actions are $fuelOff$, $fullThrottle$, and $mixtureOff$. The only restriction is that while executing this procedure, the action $fuelOn$ must not occur. In our framework, this example can be represented as follows:

> **proc** $fireOnBoard$
> $startProc(fireOnBoard)$,
> $minus([fuelOff; any^*; fullThrottle;$
> $any^*; mixtureOff],$
> $[(anyBut([fuelOn]))^*; fuelOn])$;
> $endProc(fireOnBoard)$
> **endProc**

The above examples are kept simple to illustrate how the various constructs work. The system was tested on both of the above examples, and more complicated ones. All of the above traces were generated by the implementation.

Implementation and Experimentation

Our plan recognition system was implemented using a Prolog-based version of IndiGolog, an extension of ConGolog introduced in (De Giacomo & Levesque 1999). The implementation closely follows the definitions, without any optimization for performance. The implementation assumes

that the axioms specifying the initial situation are represented as Prolog clauses and makes the closed world assumption.

The system uses a user-defined domain specification and plan library. All procedures in the library need to satisfy some restrictions. Each procedure P that is to be reflected in the scenario has to start and end with actions $startProc(P)$ and $endProc(P)$, respectively. The procedures can also use constructs $anyBut$ and $minus$.

The implementation can be used in interactive mode. Then the user is expected to enter the observed actions one by one. Also, at any point the user can issue one of the following commands: $prompt$ - list all current hypotheses, $reset$ - forget the previous actions and start fresh, and $exit$ - finish execution.

We ran some preliminary experiments using the home activities domain discussed above, with a slight modification: the last option in the plan library is now $(\pi t.[get(t), putDown(t)])$ instead of $(\pi t.get(t))$. This was done ensure that there are arbirarily long executions of the plan library. For each n, where n is the length of an observed action sequence, we randomly selected 200 sequences of n actions that could be generated by the plan library. We then ran the plan recognition system on all of those and averaged the running time. The results appear in Figure 1. We can see that our system can identify matching plans for a sequence of 80 observed actions in less than one second on average in this test domain. As well, for this domain the running time seems to grow linearly with the length of the observed action sequence.
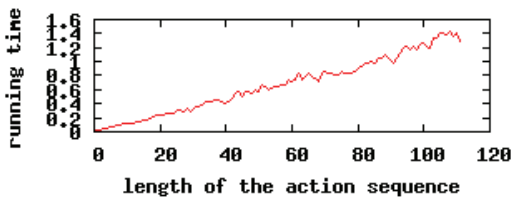

Figure 1: Average runtime (seconds) versus the length of the action sequence

Discussion

In this paper, we have described a framework for plan recognition in the Situation Calculus. The ConGolog programming language is used to specify plans. The system matches the actions of the monitored agent against the plan library and returns some scenarios, representing the execution paths that the agent may have followed.

The main differences between our account of plan recognition and the one described by (Demolombe & Hamon 2002) are that ours is able to model procedure calls within plans and that it is incremental. Because our approach to plan recognition concentrates on procedures, it is able to distinguish sub-procedures from each other as well as from top-level plans. This allows the scenarios to be fairly detailed both as to how and why a certain plan was being executed.

Because our formalism is incremental, it does not need to know the whole sequence of actions to interpret the next

step; nor does it need to re-compute matching scenarios from scratch whenever a new action is made. It would be well-suited for real-time applications or continuous monitoring.

The framework described here is easily extended with new annotations to specify, for example, the goals and preconditions of each plan and/or possible reactions to it by the monitoring system. As mentioned earlier, to fully support the recognition of concurrent executions of plans, additional annotations to track which process performed each action should be introduced. Another possible extension is to assign probabilities to actions and plans, similarly to what was done in (Demolombe & Fernandez 2005). This would make it possible to rank the possible execution hypotheses, select the most probable ones and use this to predict which actions the agent is more likely to execute next. One could also look at qualitative mechanisms for doing this. It would also be very interesting to extend our implemented plan recognition system to support some forms of incomplete knowledge about the initial situation (the implementation of IndiGolog described in (Sardina & Vassos 2005) would be a good starting point). More experimental evaluation of our system is also needed.

There has already been work on home care applications for a plan recognition system. For example, (Lin & Hsu 2006) describes a plan recognition system that includes strategies for monitoring and obtaining actions, as well as using learning to modify the plan libraries. Both of those techniques can potentially work with our system.

References

Avrahami-Zilberbrand, D., and Kaminka, G. A. 2005. Fast and complete symbolic plan recognition. In *Proc. of IJCAI-05*.

Avrahami-Zilberbrand, D., and Kaminka, G. A. 2006. Hybrid symbolic-probabilistic plan recognition: Initial stepsd. In Kaminka, G.; Pynadath, D.; and Geib, C., eds., *Modeling Others from Observations: Papers from the 2006 AAAI Workshop*.

Bui, H. H. 2003. A general model for online probabilistic plan recognition. In *Proc. of IJCAI'03*, 1309–1318.

Carberry, S. 2001. Techniques for plan recognition. *User Modeling and User-Adapted Interaction* 11(31–48).

De Giacomo, G., and Levesque, H. J. 1999. An incremental interpreter for high-level programs with sensing. In Levesque, H. J., and Pirri, F., eds., *Logical Foundations for Cognitive Agents*. Springer-Verlag. 86–102.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121:109–169.

Demolombe, R., and Fernandez, A. M. O. 2005. Intention recognition in the Situation Calculus and Probability Theory frameworks. In *Computational Logic in Multi Agent Systems*, 358–372.

Demolombe, R., and Hamon, E. 2002. What does it mean that an agent is performing a typical procedure? A formal definition in the Situation Calculus. In Castelfranci, C., and Johnson, W. L., eds., *Proceedings of the 1st International*

Joint Conference on Autonomous Agents and Multiagent Systems, 905–911. Bologne: ACM Press.

Levesque, H.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. 1997a. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31:59–84.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997b. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(59–84).

Lin, C., and Hsu, J. 2006. IPARS: Intelligent portable activity recognition system via everyday objects, human movements, and activity duration. In Kaminka, G.; Pynadath, D.; and Geib, C., eds., *Modeling Others from Observations: Papers from the 2006 AAAI Workshop*.

McCarthy, J., and Hayes, P. 1979. Some philosophical problems from the standpoint of artificial intellig ence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence*, volume 4. Edinburgh University Press. 463–502.

Plotkin, G. 1981. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Dept., Aarhus University, Denmark.

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

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

Sardina, S., and Vassos, S. 2005. The Wumpus World in IndiGolog: a preliminary report. In *6th Workshop on Nonmonotonic Reasoning, Action, and Change (at IJCAI-05)*.