

# Incremental Plan Recognition in an Agent Programming Framework

Alexandra Goultiaeva and Yves Lespérance

Dept. of Computer Science and Engineering  
York University  
Toronto, ON Canada M3J 1P3  
{cs213052, lesperan}@cs.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 ConGolog agent programming language. This provides a very rich plan specification language for the model. Our account 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 subplan relationships. We also describe an implementation of the model.

## Introduction

In the field of cognitive robotics there has been a lot of research devoted to planning and reasoning, but relatively little attention has been paid to interpreting the behavior of other agents. 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 assisting astronauts, providing assistance to people with cognitive or memory problems to allow them to live independently, etc. The system could monitor the agent's actions and provide assistance or raise alarms based on the observed actions and the state of the world.

At the heart of such a system, there needs to be a module which takes in the observed actions, and based on them selects what possible plans (or sequences of plans) the monitored agent might be performing. From that, the system could get the necessary information, and if needed, react accordingly.

There has been a lot of work in the area of plan recognition, e.g. (Kautz 1991). Here, we develop an approach that supports very rich plans and can be integrated in a cognitive agent programming framework.

In (Demolombe & Hamon 2002), the authors make the distinction between procedures as executed by a human and programs for machines. Such procedures are descriptions of plans that can be used to recognize observed actions; they may leave some actions unspecified, and may introduce constraints on the actions that can be performed. Then, the authors provide a formal definition of what it means for an

agent to be performing a procedure. The definition is based on the Golog agent programming language and its semantics. Plans are represented as Golog programs, with two additional constructs:  $\sigma$ , which matches any sequence of actions, and  $\alpha_1 - \alpha_2$ , which matches an execution of plan  $\alpha_1$  as long as it does not also match an execution of  $\alpha_2$ . The paper defines a predicate  $Do_p(\alpha, s, s')$ , which means that the plan  $\alpha$  matches the observed actions between  $s$  and  $s'$ .

Here, we provide an alternative formalization and implementation of the plan recognition framework. 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.

A 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.

To achieve this, plan recognition is modeled in terms of single step transitions in the style of (Plotkin 1981). The main predicate that is defined is  $nTrans(\delta, s, s_a, \delta', do(a, s), s'_a)$ , which means that in one observed transition, which can match the next step of procedure  $\delta$ , it is possible to move from situation  $s$  to situation  $do(a, s)$ . In addition,  $\delta'$  is what remains of procedure  $\delta$  after the execution of that action. The situations  $s_a$  and  $s'_a$  are annotated situations before and after the action, respectively, which include information about which procedures have started and completed. The details are described later in the paper.

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 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*, namely that situation in which no actions have yet occurred; 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 the 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  $\Sigma$  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), an extension of Golog (Levesque *et al.* 1997a), providing a rich set of programming constructs as follows:

$\alpha$ ,	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
$\delta^*$ ,	nondeterministic iteration
<b>if</b> $\phi$ <b>then</b> $\delta_1$ <b>else</b> $\delta_2$ <b>endIf</b> ,	conditional
<b>while</b> $\phi$ <b>do</b> $\delta$ <b>endWhile</b> ,	while loop
$\delta_1 \parallel \delta_2$ ,	concurrency with equal priority
$\delta_1 \gg \delta_2$ ,	concurrency with $\delta_1$ at a higher priority
$\delta^\parallel$ ,	concurrent iteration
$\langle \phi \rightarrow \delta \rangle$ ,	interrupt
$p(\theta)$ ,	procedure call

Among these constructs, we notice the presence of of non-deterministic constructs. These include  $(\delta_1 \mid \delta_2)$ , which nondeterministically chooses between programs  $\delta_1$  and  $\delta_2$ ,  $\pi x. \delta$ , which nondeterministically picks a binding for the

variable  $x$  and performs the program  $\delta$  for this binding of  $x$ , and  $\delta^*$ , which performs  $\delta$  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  $\delta_1$  and  $\delta_2$ . We refer the reader to (De Giacomo, Lespérance, & Levesque 2000) for a detailed account of ConGolog.

Golog and its successors have been used to build a number of applications. These include a demonstration mail delivery robot at Toronto and York Universities, a robot museum guide at University of Bonn (Burgard *et al.* 1998), a character specification program for computer animation (Funge 1998), and others.

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  $\delta$  starting in situation  $s$ , one can get to situation  $s'$  in one elementary step with the program  $\delta'$  remaining to be executed.  $Final(\delta, s)$  means that program  $\delta$  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  $\delta$  (a sequence of transitions) that starts in situation  $s$  and terminates in situation  $s'$ :

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

where  $Trans^*$  is the reflexive transitive closure of  $Trans$ . An offline execution of  $\delta$  from  $s$  is a sequence of actions  $a_1, \dots, a_n$  such that:  $\mathcal{D} \cup \mathcal{C} \models Do(\delta, s, do(a_n, \dots, do(a_1, s) \dots))$ , 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. Depending on the application, the plan library may also include faulty plans to allow the system to recognize common errors in plans and offer advice for correcting them. 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. However, when there is concurrency between procedures in the plan library, the annotated situation as currently defined is no longer sufficient to determine which procedure an observed action belongs to. Additional annotations would have to be used to disambiguate this. Concurrency can also lead to a huge increase in the number of plan execution hypotheses, so control mechanisms would be needed. Thus, we do not consider the case of concurrency between procedures in the plan library in this paper and leave the problem for future work.

After the inclusion of the annotation actions, for each sequence of observable 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.

As mentioned in (Demolombe & Hamon 2002), in many situations it would be useful for the procedures to leave some actions unspecified, or to place additional constraints on the plans. So, two new constructs, discussed in more detail later, are introduced. 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 to write 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 how the condition was achieved.

The second construct is  $minus(\delta, \hat{\delta})$ . This construct matches any execution that would match  $\delta$ , as long as that execution 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  $minus(cleanUp, brushTeeth)$ .

The first predicate defined for the new formalism is  $aTrans$ . It is, in essence, a form of  $Trans$  which only allows the next step to be an annotation action or a test action. Essentially,  $aTrans$  means a transition step that cannot be observed. The helper predicate  $Annt$  is true if and only if the action passed to it is an annotation action.

$$\begin{aligned} Annt(a) &\stackrel{\text{def}}{=} \exists n. a = startProc(n) \vee \exists n. a = endProc(n) \\ aTrans(\delta, s, \delta', s') &\stackrel{\text{def}}{=} \\ &Trans(\delta, s, \delta', s') \wedge \\ &(\exists a. (s' = do(a, s) \wedge Annt(a)) \vee s' = s) \end{aligned}$$

The predicate  $rTrans$  is a form of  $Trans$  which only allows observable actions:

$$\begin{aligned} rTrans(\delta, s, \delta', s') &\stackrel{\text{def}}{=} \\ &Trans(\delta, s, \delta', s') \wedge \exists a. s' = do(a, s) \wedge \neg Annt(a) \end{aligned}$$

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

$$aTrans^*(\delta, s, \delta', s') \stackrel{\text{def}}{=} \forall R. [\dots \supset R(\delta, s, \delta', s')]$$

where  $\dots$  stands for the universal closure of the following:

$$\begin{aligned} &R(\delta, s, \delta, s) \\ &R(\delta, s, \delta', s') \wedge aTrans(\delta', s', \delta'', s'') \supset R(\delta, s, \delta'', s'') \end{aligned}$$

The transition predicate  $nTrans(\delta, s_r, s_a, \delta', s'_r, s'_a)$  is the main predicate in our plan recognition framework. It is defined so that  $\delta'$  is the program remaining from  $\delta$  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 that. 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 that. Effectively, our definition below amounts to  $nTrans$  being equivalent to  $aTrans^*$  composed with  $rTrans$ :

$$\begin{aligned} nTrans(\delta, s_r, s_a, \delta', s'_r, s'_a) &\stackrel{\text{def}}{=} \\ &(\exists \delta'', s''_a, a). aTrans^*(\delta, s_a, \delta'', s''_a) \\ &\wedge rTrans(\delta'', s_r, \delta', do(a, s_r)) \\ &\wedge s'_r = do(a, s_r) \wedge s'_a = do(a, s''_a). \end{aligned}$$

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:

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

The construct  $anyBut([a_1, \dots, a_n])$ , mentioned before, matches any action except for those in the list. It is simply an abbreviation, included for convenience. It is defined as follows:

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

Another shorthand construct,  $any$ , can be defined to match any action without exceptions:

$$any \stackrel{\text{def}}{=} anyBut([])$$

The construct  $minus(\delta, \hat{\delta})$  is used to constrain the execution of a procedure in terms of the actions that can be performed. An execution should match this construct only if it matches  $\delta$ , but does not match  $\hat{\delta}$ . We need to define what a step of execution for this construct is, and the remaining program. Also, note that  $\hat{\delta}$  has to match all observable actions, but might have different annotation and test actions; those differences should be ignored.

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

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

This says the following: if the next step of the plan  $\delta$  is not an observable action, then the remaining program is what remains of  $\delta$  minus  $\hat{\delta}$ ; if  $\delta$  performs an observable action, and  $\hat{\delta}$  cannot match that action, then the remaining program is what remains of  $\delta$ ; if  $\hat{\delta}$  can match the observable action performed by  $\delta$  but it is not final, then the remaining program is what remains of  $\delta$  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  $minus$ . Note also that recursive procedures can be handled as in (De Giacomo, Lespérance, & Levesque 2000).

Also, an additional axiom is added to specify  $Final$ . The construct  $minus$  is considered finished when the main procedure is finished, but the exception procedure is not:

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

We use  $\mathcal{C}'$  to denote the extended ConGolog axioms:  $\mathcal{C}$  together with the above two.

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

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

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.<sup>1</sup>

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 smallest relation that includes the identity relation and is closed under  $nTrans$ :

$$\begin{aligned} nTrans^*(\delta, s_r, s_a, \delta', s'_r, s'_a) \stackrel{\text{def}}{=} & \\ \forall R. [\dots \supset R(\delta, s_r, s_a, \delta', s'_r, s'_a)] & \end{aligned}$$

where  $\dots$  stands for the universal closure of the following:

$$\begin{aligned} R(\delta, s_r, s_a, \delta, s_r, s_a) & \\ R(\delta, s_r, s_a, \delta', s'_r, s'_a) \wedge nTrans(\delta', s'_r, s'_a, \delta'', s''_r, s''_a) & \\ \supset R(\delta, s_r, s_a, \delta'', s''_r, s''_a). & \end{aligned}$$

The predicate  $allTrans(s_r, s_a, \delta_{rem})$  means that  $s_a$  denotes a possible annotated situation that matches the situation  $s_r$ , and  $\delta_{rem}$  is the remaining plan. It can be defined as follows:

$$\begin{aligned} allTrans(s_r, s_a, \delta_{rem}) \stackrel{\text{def}}{=} & \\ nTrans^*(planLibrary, S_0, S_0, \delta_{rem}, s_r, s_a) & \end{aligned}$$

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

The set of all the remaining programs and their corresponding annotated situations for a given real situation can be defined as follows:

<sup>1</sup>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.  $\pi n. (PositiveInteger(n)?; a; b(n))$ .

$$allPlans(S) \stackrel{\text{def}}{=} \{(\delta, S_a) \mid \mathcal{D} \cup \mathcal{C}' \models allTrans(S, S_a, \delta)\}$$

As mentioned earlier, the system also allows incremental calculation of the set of plans that the agent may be executing. It is straightforward to show that:

$$allPlans(do(A, S)) = \{(\delta, S_a) \mid (\delta', S'_a) \in allPlans(S) \wedge \mathcal{D} \cup \mathcal{C}' \models nTrans(\delta', S, S'_a, \delta, do(A, S), S_a)\}$$

Note that according to the definition of  $nTrans$ , the program would stop matching the actions against the plan as soon as the last real action was matched. This means that if the procedure was finished, it would be recognized as finished only after the next action has occurred (because before that, the action  $endProc(proc)$  would not have been reached). In some cases this could be viewed as strange. However, in many situations the next action must be known before deciding whether or not a procedure is finished. For example, consider a program defined as follows:

```
proc p1 startProc(p1); a*; endProc(p1)endProc
```

If the library has a plan  $p1$ ;  $anyBut([a])$ , then it is impossible to know whether or not procedure  $p1$  is finished, until the next action is seen. If the next action is  $a$ , that means that  $p1$  is not yet finished. If it is an action other than  $a$ , then that means that  $p1$  is finished.

If it is important to check whether the plan has finished or not, it is easy to do this within the current framework. We can insert a new special action  $flush$  at the end of the plan library. Then if we need to find out whether a plan is finished or not, we can check to see if it can match the action  $flush$  as its next transition.

## Examples

The main example described here is a simulation of activities in a home. There are four rooms: bedroom, kitchen, living room and a bathroom. There are also four objects: a toothbrush, a book, a spoon, and a 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.

There are four possible primitive actions:

- $goTo(room)$ : changes the location of the agent; it is assumed that the agent can get from any room to any other room in one action;
- $pickUp(object)$ : only possible if the agent is in the same room as the object; this causes the object to be “in hand”;
- $putDown(object)$ : 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.

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. 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)); (Hold(t) = false)?; getTo(loc(t));
  pickUp(t); endProc(get(t))
endProc;
```

```
proc putAway(t)
  startProc(putAway(t)); (Hold(t) = true)?;
  (InPlace(t, room)?)?; getTo(room); 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;
```

Note that all procedures are bracketed with annotation actions. Their start and end will be documented in the annotated situation.

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.

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

```

proc cleanUp
  startProc(cleanUp);
  while( $\exists t. \neg \text{InPlace}(t)$ )do
     $\pi$  thing.
       $\neg \text{InPlace}(\text{thing}, \text{loc}(\text{thing}))?$ ;
      (get(thing)|putAway(thing))
  endWhile;
  endProc(cleanUp)
endProc

```

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

```

proc library
  (cleanUp|(brushTeeth|(readBook|( $\pi t. \text{get}(t)$ ))))*.
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) - not finished
  goTo(kitchen)
-----
proc get(cup) - not finished
  goTo(kitchen)
-----
proc get(spoon) - not finished
  goTo(kitchen)
-----
proc readBook - not finished
  proc get(book) - not finished
    goTo(kitchen)
-----
proc cleanUp - not finished
  proc get(book) - not finished
    goTo(kitchen)

```

The system is trying to guess what the user is doing by going to the kitchen. There are five plans in the library that might have this as a first action: *get(book)*, *get(cup)*, *get(spoon)*, *readBook* and *cleanUp*.

Note that the possibilities of doing *cleanUp* by getting a cup or a spoon are not listed. That is because both spoon and cup are already in their places, so if the agent picked them up, it would not constitute 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) - not finished
  goTo(kitchen)
  pickUp(book)
-----

```

```

proc readBook - not finished
  proc get(book) - not finished
    goTo(kitchen)
    pickUp(book)
-----
proc cleanUp - not finished
  proc get(book) - not finished
    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 - not finished
  proc get(book) - finished
    goTo(kitchen)
    pickUp(book)
  use(book)

```

Now, let us consider a different scenario. In order to demonstrate the *minus* and *anyBut* constructs, two variants of *cleanUp* have been defined. In the first one, an arbitrary action is allowed at the end of every iteration of the loop:

```

proc cleanUpu
  startProc(cleanUpu);
  while( $\exists t. \neg \text{InPlace}(t)$ )do
     $\pi$  thing.  $\neg \text{InPlace}(\text{thing}, \text{loc}(\text{thing}))?$ ;
    (get(thing)|putAway(thing)); (any|nil)
  endWhile;
  endProc(cleanUpu)
endProc

```

The second one, 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 cleanUpm
  startProc(cleanUpm);
  minus(
    while( $\exists t. \neg \text{InPlace}(t)$ )do
       $\pi$  thing.  $\neg \text{InPlace}(\text{thing}, \text{loc}(\text{thing}))?$ ;
      (get(thing)|putAway(thing)); (any|nil);
    endWhile,
    [brushTeeth]);
  endProc(cleanUpm)
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 - not finished
  proc get(toothbrush) - not finished
    goTo(livingRoom)
    pickUp(toothbrush)

```

where  $k$  is either no subscript, a subscript  $u$  or a subscript  $m$ , depending on the version of the program used.

Now suppose that the next action is  $use(toothbrush)$ . The original version of  $cleanUp$  cannot match the observed action, so there would be no remaining hypotheses.

However, 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 - not finished
  proc get(toothbrush) - finished
    goTo(livingRoom)
    pickUp(toothbrush)
    use(toothbrush)
  proc putAway(toothbrush) -
                                not finished
    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. On the other hand,  $cleanUp_u$ , which is identical to  $cleanUp_m$  except for the minus construct, matches that action, and produces the following scenario:

```
proc cleanUp_u - not finished
  proc get(toothbrush) - finished
    goTo(livingRoom)
    pickUp(toothbrush)
    use(toothbrush)
  proc putAway(toothbrush) -
                                not finished
    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 appear.

In our framework, this example can be represented in two ways:

```
proc fireOnBoard1
  startProc(fireOnBoard1),
  minus([fuelOff; any*; fullThrottle;
        any*; mixtureOff],
        [(anyBut([fuelOn]))*; fuelOn]);
  endProc(fireOnBoard1)
endProc
```

The above representation uses the  $minus$  construct to forbid any sequence of actions that involves the action  $fuelOn$ .

The alternative definition below uses the exception list in the  $anyBut$  construct to allow any action except for  $fuelOn$ .

```
proc fireOnBoard2
  startProc(fireOnBoard2);
  fuelOff; (anyBut([fuelOn]))*; fullThrottle;
  (anyBut([fuelOn]))*; mixtureOff,
  endProc(fireOnBoard2)
endProc
```

For this example, both definitions are equivalent. The action sequence  $[fuelOff, fullThrottle, mixtureOff]$  would match both examples, but the sequence  $[fuelOff, fullThrottle, fuelOn]$  would match neither.

In general, the difference between the  $anyBut$  and  $minus$  constructs is that  $minus$  can rule out any execution of a program, while  $anyBut$  can only rule out a single primitive action specified in the list.

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

Our plan recognition system was implemented using the possible-values version of IndiGolog described in (Sardina & Vassos 2005). The implementation was done in Prolog as an extension to this IndiGolog implementation. IndiGolog (De Giacomo & Levesque 1999) is an extension of ConGolog that supports incremental planning and plan execution.

In order to run, the system needs a user-defined domain specification and the plan library. The following needs to be specified:

- An IndiGolog definition of the domain: actions, fluents, preconditions, effects, and initial state. This is defined using the standard IndiGolog syntax.
- A predicate  $clientAction(A)$ , which is true for any primitive action  $A$  that the observed agent may do.
- A procedure called  $planLibrary$ , which is a collection of calls to all procedures in the plan library, and serves as the starting point for matching observed actions.

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; no concurrency is allowed for this version. The procedures can also use constructs  $anyBut$  and  $minus$ .

The system uses IndiGolog's interactivity to implement the incremental plan recognition. 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 the current hypotheses; this command would cause the program to output the possible annotated situations as trees, and their corresponding remaining programs.
- *reset*: forget the previous actions and start fresh from the plan library.
- *exit*: finish execution.

## Discussion

In this paper, we have described a framework for plan recognition implemented in IndiGolog. 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, for example, the goals and preconditions of each plan. This would add more information to the recognized scenario.

It is assumed that all possible plans are in the plan library, and are closely followed. Some leeway can be built into the plans using the *anyBut* action, for example, but this comes at the cost of how detailed the resulting scenario is. It is possible to define any plan as *any\**, but then the resulting scenario would not be very useful. A large plan library and domain definition should be able to solve some of those problems. In future work, we will consider ways to make the system more robust and have it detect partial matches.

The current plan recognition system does only that, recognize plans. To be directly useful in applications, it would need additional components working together with plan recognition. These could include a system of sensors (which would provide the information about actions), a component to analyze the matching plan executions, a component to decide when and how to respond to the monitored agent's actions, and an interface to the real world: communication with the agent, ability to call for help, etc.

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.

## References

Burgard, W.; Cremers, A. B.; Fox, D.; Hahnel, D.; Lake-meyer, G.; Schulz, D.; Steiner, W.; and Thrun, S. 1998. The interactive museum tour-guide robot. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, 11–18.

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. Bologna: ACM Press.

Funge, J. 1998. *Making Them Behave: Cognitive Models for Computer Animation*. Ph.D. Dissertation, University of Toronto, Toronto, Canada.

Kautz, H. A. 1991. A formal theory of plan recognition and its implementation. In Allen, J. F.; Kautz, H. A.; Pelavin, R.; and Tenenber, J., eds., *Reasoning About Plans*. San Mateo (CA), USA: Morgan Kaufmann Publishers. 69–125.

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).

McCarthy, J., and Hayes, P. 1979. Some philosophical problems from the standpoint of artificial intelligence. 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)*.