Analysis of Agent Programs using Action Models

José Ramírez and Hector Geffner

Dpto. de Computación
Universidad Simón Bolívar
Apartado 89000, Caracas 1080-A, Venezuela
{jramire, hector}@usb.ve
http://www.ai.usb.ve

Abstract. The use of action models for the analysis of agent control programs can be useful for two reasons. First, it promises to deliver better tools for the simulation, verification and synthesis of control programs, and second it presents challenging problems for theories of action and knowledge. In this paper we use a theory of actions and knowledge developed elsewhere to analyze control programs for robot navigation tasks. We model both physical and sensing actions and establish conditions under which different control programs are executable and lead the agent to the intended goal.

Topics: Temporal and spatial reasoning, Knowledge Representation, Agent-oriented programming, Robotics

Analysis of Agent Programs using Action Models

Abstract. The use of action models for the analysis of agent control programs can be useful for two reasons. First, it promises to deliver better tools for the simulation, verification and synthesis of control programs, and second it presents challenging problems for theories of action and knowledge. In this paper we use a theory of actions and knowledge developed elsewhere to analyze control programs for robot navigation tasks. We model both physical and sensing actions and establish conditions under which different control programs are executable and lead the agent to the intended goal.

Topics: Temporal and spatial reasoning, Knowledge Representation, Agent-oriented programming, Robotics

1 An example

Consider a rectangular environment with an agent trying to reach a goal object (Figure 1). The agent has a position and orientation (displayed by the dot within the circle that represents the agent) and can either move forward or rotate. He has also sensing capabilities that allow him to determine the position of the target object within certain constraints: e.g., that no other object is on the way, that the distance or angle to the goal object is within his reach, etc.

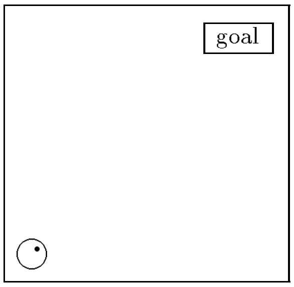

Fig. 1. A simple rectangular world with an agent and a goal

Intuitively if these constraints are met, a simple control loop expressed as the sequence of condition-action pairs in Program 1 should lead the agent to the goal. Methodologies for building agent programs of this form have been proposed by Brooks [3], Nilsson [12] and others. Such programs are normally tested on simulated worlds or in the real world; but that testing does not assure that in all situations the programs will behave well, or even, if they are executables. Here we aim to show that such programs can also be tested over sufficiently rich action models.

Program 1 Rotate and Move to Goal

$$reached(goal) \rightarrow done$$
$$see(goal) \wedge facing(goal) \rightarrow move$$
$$see(goal) \rightarrow rotate$$

An action model is a description of the effects of actions on both the environment (e.g., [6]) and the agent's internal state or knowledge [10, 13]. Since a control program can be seen a mapping from knowledge to actions, an action model can be used to predict whether a control program is executable and whether it will lead the agent to its intended goal (see also [9, 2]).

An action model is a good standpoint for the evaluation of the executability of a program, determining if the constraints associated with the actions are met, the conditions for the execution of the actions are achieved, and if such conditions can be evaluated. In some agent programming approaches the executability is guaranteed by imposing restrictions on the evaluation mechanism [12], establishing consistency rules [15] or providing mechanisms of detection of abnormal situations and fallback [3]. The verification that the program will lead the agent to its goal is also possible given an accurate action model that support the assumption that the goal condition is reachable and that the actions performed will eventually make **true** that condition.

The use of action models for the analysis of agent control programs can be useful for two other reasons: first, it may eventually deliver better tools for the simulation, optimization and synthesis of control programs, and second, it presents challenging problems for current theories of action. Indeed, the simple program above involves both physical and sensing actions, yet few theories exist that can make sense of plans in such settings.

2 Action Theories

The model for actions below is a variation of a model reported in [5], which in turn is an extension of [6]. It comprises a *language* for describing actions in the form of a logical theory, and a *semantics* that maps such theories into sets of state trajectories s_0, s_1, s_2, ... where each s_i represents the state of the world at time i. The modeling language is built up from constant, function and predicate symbols distinguished by certain semantic properties:

1. *fixed symbols* have denotations that are fixed and *known* across time
2. *fluent symbols* have denotations that tend to persist
3. *action symbols* denote actions

Fixed symbols include symbols like '3', '+', '=', '$\surd$', etc., whose denotation is fixed and standard, and other symbols that we call *identifiers* that we regard as self-denoting.

For simplicity we assume that action symbols are *propositional symbols* denoting *actions,* and fluents are either constant or function symbols of arity one. Thus a relation like $\mathsf{on}(x, y)$ in the blocks world can be modeled by an equality like $\mathsf{loc}(x) = y$. Moreover, each functional fluent f will have a *type* of the form $D_f \rightarrow R_f$ meaning that the function denoted by f takes elements from D_f and maps them into R_f. The fluent loc for example may have a type $BLOCKS \rightarrow R$, where $BLOCKS$ is a set of block identifiers ($block\_A$, $block\_B$, etc) and R is the real line. The domain D_f of the functions denoted by functional fluents is assumed to be given by a set of identifiers.

Terms, atoms, and formulas are defined from the constant, function, and predicate symbols in the standard way, except that the only terms of the form $f(t)$ when f is a fluent are the ones in which t belongs to D_f. Such terms, as well as the constant fluent symbols, are called *fluent terms.*

Action Rules The *action rules* are rules of the form

$$A \wedge C \rightarrow L$$

where A is an *action*, C is a formula not involving action symbols, and L is an *assignment* of the form $F := t$, where F is a fluent term and t is a term. For example, the action rules:

$$hit \wedge \mathsf{h} > \delta \rightarrow \mathsf{h} := \mathsf{h} - \delta$$
$$hit \wedge \mathsf{h} \leq \delta \rightarrow \mathsf{h} := 0$$

say that after hitting a nail with a hammer, its distance h to the wall decreases by a fixed constant δ if its original distance was greater than δ, and to zero otherwise.

Definitions

New *terms* and *atoms* can be introduced in the language by means of *definitions.* For example, the atom $\mathsf{close}(block\_A)$ can be defined to be true just when $|\mathsf{pos}(block\_A)| \leq 0.1$ is true, and similarly the term $\mathsf{dist}(block\_A)$ can be defined as the value of the term $\mathsf{pos}(block\_A) - \mathsf{pos}(me)$. New terms t are defined by expressions of the form:

$$t \stackrel{\text{def}}{=} t'$$

meaning that the denotation (value) of t is the denotation of the term t'. Similarly, new atoms p can be defined either by an expression of the form:

$$p \stackrel{\text{def}}{=} A$$

meaning that the denotation of p is the denotation of A, or by collections of clauses of the form

$$p \ \textbf{if} \ A$$

meaning that p is true *only if* some such clause A is true (i.e., such partial definition is completed assuming that p is *false* by default). Defined terms and atoms can be used anywhere as long as they do not introduce circularities in the definitions.

An *action theory* T is a triplet of the form $\langle D, A, O\rangle$, where D is a *domain theory* containing action rules and definitions, A is set of timed actions, and O is a set of observations. A timed action is an expression of the form $p[i]$, where p is an action symbol and i is a time point (a non-negative integer). An observation is an expression of the form $F[i]$, where F is a non-action formula and i is a time point.

For example, the rules above about hitting a nail together with the actions $A = \{hit[0], hit[1], hit[2]\}$ and the observation $\mathsf{h}[0] = 5$ constitute an action theory. The semantics of such theories is given below. Provided that the value of the fixed symbol δ is 2, this action theory for instance yields the conclusion $\mathsf{h}[3] = 0$.

3 Semantics

The semantics maps an action theory into a set of state trajectories s_0, s_1, ... where s_i stands for the *state* of the world at time i.

3.1 States

Each state s is an *interpretation* over a domain D that assigns a denotation x^s to each expression x in the language in the standard way:

–$c^s \in D$, if c is a constant symbol
–$f^s \in D^n \rightarrow D$, if f is a function symbol with arity n
–$[f(\mathbf{t})]^s = f^s(\mathbf{t}^s)$
–...

For standard symbols x (numerals, arithmetic operators and predicates, etc.), x^s is the standard denotation of x, while for identifiers $x^s = x$. For terms t defined as $t \stackrel{def}{=} t'$, t^s is defined as t'^s, while for atoms p defined as $p \stackrel{def}{=} A$, p^s is defined as A^s. Similarly, for atoms p defined by a collection of clauses of the form p **if** A, p^s is true iff A^s is true for some such clause. A state s *satisfies* a formula F, written $s \models F$, if $F^s = \mathbf{true}$.

3.2 Trajectories

A trajectory t is a sequence of states s_0, s_1, ..., s_i, ... over a common domain D. A trajectory s_0, s_1, ... is *admissible* relative to a given *domain theory* if the changes in every transition s_i to s_{i+1}, for $i \geq 0$, are supported by the action rules; i.e.,

–$f^{s_{i+1}} = t^{s_i}$ if $f := t$ is supported in s_i, else $f^{s_{i+1}} = f^{s_i}$
–$f^{s_{i+1}}(c) = t^{s_i}$ if $f(c) := t$ is supported in s_i, else $f^{s_{i+1}}(c) = f^{s_i}(c)$ for each $c \in D_f$.

An assignment $f(t) := t'$ (f :=t) is supported in a state s_i when for some rule $F \rightarrow f(t) := t$ ($F \rightarrow f := t$) its antecedent F is true in s_i.

3.3 Models

The *models* of an action theory $T = \langle D, A, O \rangle$ are the admissible trajectories (relative to D) that are *compatible* with both the actions A and the observations O.

A trajectory $s_0, s_1, \ldots$ is *compatible* with the observations if for each expression $F[i] \in O$, F^{s_i} is true, and is *compatible with the actions* A iff for each action symbol p, p^{s_i} is true iff $p[i] \in A$ (i.e., actions not in A are assumed to be false). For the example above, it is simple to check that a transition from a state s_i to a state s_{i+1} is admissible if:

–$\mathsf{h}^{s_{i+1}} = \mathsf{h}^{s_i}$, when hit^{s_i} is false or $\mathsf{h}^{s_i} = 0$,
–$\mathsf{h}^{s_{i+1}} = 0$, when hit^{s_i} is true and $\mathsf{h}^{s_i} \leq \delta$, or $\mathsf{h}^{s_i} = 0$
–$\mathsf{h}^{s_{i+1}} = \mathsf{h}^{s_i} - \delta$, when $\mathsf{h}^{s_i} > \delta$ and hit^{s_i} is true

In the resulting models $s_0, s_1, \ldots$, of the theory, if $\mathsf{h}^{s_0} = 5$ then $\mathsf{h}^{s_1} = 3$, $\mathsf{h}^{s_2} = 1$, and $\mathsf{h}^{s_3} = 0$.

4 Control Theories

Control theories are similar to action theories except that actions are replaced by control programs. In other words, a control theory has the form $C = \langle D, P, O \rangle$, where D and O are as above, but actions A are replaced by a control program P. A control program is a finite sequence of condition-action pairs:

$$c_1 \rightarrow a_1 \;;\; c_2 \rightarrow a_2 \;;\; c_3 \rightarrow a_3 \;;\; \ldots$$

where each a_i is an action and each c_i is a formula which does not involve any actions. That sequence is evaluated from scratch at every time point $i \geq 0$, and the first action whose condition is true is executed. Later on we will consider two conditions such programs must satisfy: namely, the agent must have the *knowledge* to evaluate the conditions c_i, and if c_i is the first condition that evaluates to true, the *preconditions* of a_i must be true as well.

The *models* of a control theory $T = \langle D, P, O \rangle$ are the admissible trajectories (relative to D) that are *compatible* with both the observations O and the program P where:

Definition 1.A trajectory $s_0, s_1, \ldots$ is compatible with program P if for each state s_i and each action a, a is true in s_i if and only if $a = a_j$ and $c_j \rightarrow a_j$ is the first condition-action pair in P whose condition c_j is true in s_i.

Going back to the example above, it is simple to check that the effect of the actions $hit[0]$, $hit[1]$ and $hit[2]$ can be achieved by the simple program

$$\mathsf{h} > 0 \rightarrow hit$$

Moreover, the program will achieve the effect $\mathsf{h} = 0$ for *any* positive initial value of h as long as it can evaluate the condition $\mathsf{h} > 0$ (see below).

5 Action Constraints

So far we have ignored that actions often have preconditions which may prevent the action to be executed. For example, a agent cannot move forward when blocked by an obstacle, he cannot pick up an object if he does not have an empty hand, etc.

We accommodate preconditions by extending action and control theories with a fourth component: *action constraints*. Action constraints are expressed by formulas of the form:

$$a \supset C$$

where a is an action, and C is a formula expressing a *precondition* for a.

The semantics of action constraints is very simple and follows the STRIPS model [4]: an action theory is *executable* when no model of the theory contains a state s_i that violates an action constraint. For example, if $move \supset \mathsf{clear\_front}$ is a constraint, a theory in which a *move* is performed when $\mathsf{clear\_front}$ is false is *non-executable*.

6 Knowledge

The executability of control theories is a bit more subtle than the executability of simple actions as the agent has to be able to *evaluate* the conditions in the program. A condition-action pair may be $facing(goal) \rightarrow move$, yet the agent may not *know* whether it is actually facing the goal or not (say because of limited visibility, presence of obstacles, etc). In such case the program can not be executed.

In order to characterize the executability of programs we need to model what the agent *knows* and therefore what conditions the agent can evaluate. For that purpose we introduce an intensional operator K so that $K(x)$ for a term or (objective) formula x means that *the denotation of x is known* (see [5] for details, and [10] and [13] for related approaches). The knowledge of the agent will be grounded on the definitions in the theory and on the expressions x whose denotations are *observable* (e.g., the time of the day is observable when looking at the clock).[1] We express that the denotation of a term or formula x is observable by writing $obs(x)$. The conditions C that make certain terms or expressions x observable are encoded by means of defining clauses of the form (see Section 2):

$$obs(x) \ \ \mathbf{if} \ \ C$$

that indicate that $obs(x)$ is true when some such formula C is true. We say that an expression x is observable in a state s if $obs(x)$ is true in s.

The truth of the epistemic expression $K(x)$ in a state s is determined by the denotation of x in all states s' that are possible from s given the definitions and expressions y of what is observable in s:[2]

Definition 2. $K(x)$ is true in s iff $x^s = x^{s'}$ for all states s' that are *accessible* from s, where s' is *accessible* from s if for all expressions y observable in s, $y^{s'} = y^s$.

Using again the example of hitting a nail with a hammer, if the atom *high* is defined in the theory as $\mathsf{h} > 10$, the truth of the atom *high* will be *known* if the fluent h is *observable* (i.e., $K(high)$ will follow from $obs(\mathsf{h})$). This is because for

[1] The model of knowledge below is a simplification of the model in [5] which assumes that the agent knows the rules in the theory and also has memory about past observations.

[2] Notice that all states automatically satisfy the definitions.

all states s' accessible from s we must have $\mathsf{h}^{s'} = \mathsf{h}^s$ (because h is observable), and hence $(\mathsf{h} > 10)^{s'} iff (\mathsf{h} > 10)^s$.

More generally all observable expressions are always *known,* and if an expression x (atom or term) is defined in terms of expressions y that are known, x will be known as well. Provided with this model of knowledge, the *executability* of a program relative to a given domain theory can be characterized as follows:

Definition 3. A program is executable if in *every* state s_i of every model s_0, s_1, ... 1) all action constraints are satisfied, and 2) all conditions c_0, c_1, ..., c_j up to and including the first condition that is true in s_i are known.

In all programs that we consider, the action a_1 associated with the first condition c_1 will be the special action **done** that has no effects. We will be able to say that a program P *achieves* a formula (goal) G if in every model the action **done** becomes true at some point, and at such point G is true.

7 Analysis of Simple Navigation Programs

7.1 Situation 1

We are ready to analyze the Program 2, an extended version of the program discussed in Section 1. For that we need to model each of the conditions and actions involved in the program.

Program 2 Amble and Move to Goal

$$\begin{aligned} reached(goal) &\rightarrow done \\ see(goal) \wedge facing(goal) &\rightarrow move \\ see(goal) &\rightarrow rotate \\ \neg blocked &\rightarrow move \\ \mathbf{true} &\rightarrow rotate \end{aligned}$$

First we assume a set of object identifiers OBJ, containing the identifier *goal* for the goal object, and a coordinate system with origin in the bottom-left corner of the rectangular environment shown in Figure 1. The position and orientation of the agent will be represented by three fluents xpos, ypos and angle. Following the denotation suggested by Latombe in [7] and assuming that the agent is a rigid *free-flying* object, xpos and ypos are the coordinates of the center of the agent with respect to the origin of environment, and angle is the angle between the x-axis of the agent and the environment, restricted to the interval $(0, 2\pi]$ with modulo 2π arithmetic. The effect of the two actions on these fluents is captured by the rules:

$$\begin{aligned} rotate &\rightarrow \mathsf{angle} := \mathsf{angle} + \delta angle \\ move &\rightarrow \mathsf{xpos} := \mathsf{xpos} + \delta dist \cos(\mathsf{angle}) \\ move &\rightarrow \mathsf{ypos} := \mathsf{ypos} + \delta dist \sin(\mathsf{angle}) \end{aligned}$$

where $\delta angle$ and $\delta dist$ are two known constants standing for the angular and linear step sizes.

The agent has no information about the absolute location of the objects (or itself), yet it can determine the *relative* positions $dist(obj)$ and angles $angle(obj)$ of the objects obj he can 'see' (this is the so-called *indexical* information [8]). Given that the *absolute* position of the object obj is captured by the fluents $\mathsf{xloc(obj)}$ and $\mathsf{yloc(obj)}$, their position and angles relative to the agent can be defined as follows:

$$\begin{aligned} dist(obj) &\stackrel{\text{def}}{=} \sqrt{(\mathsf{xpos} - \mathsf{xloc}(obj))^2 + (\mathsf{ypos} - \mathsf{yloc}(obj))^2} \\ angle(obj) &\stackrel{\text{def}}{=} \mathsf{angle} - \theta(obj) \\ \theta(obj) &\stackrel{\text{def}}{=} \sin^{-1} \frac{\mathsf{ypos} - \mathsf{yloc}(obj)}{dist(obj)} \end{aligned}$$

where obj stands for each of the identifiers in the set OBJ.

With these definitions, some of the conditions in the program can be modeled by the schemas:

$$\begin{aligned} facing(obj) &\stackrel{\text{def}}{=} angle(obj) \leq \delta angle \\ reached(obj) &\stackrel{\text{def}}{=} facing(obj) \wedge close(obj) \\ close(obj) &\stackrel{\text{def}}{=} dist(obj) \leq \delta dist \end{aligned}$$

The condition $see(goal)$ in the program is special because it aims to model the *perceptual* machinery of the agent. If we want to predict how the agent is going to behave, we have to provide a model for the condition *see*.[3] A very simple model assumes that the agent is going to *see* any object that is within a certain maximal distance ($vdist$) and a certain maximal angle ($vangle$):

$$see(obj) \stackrel{\text{def}}{=} dist(obj) \leq vdist \wedge angle(obj) \leq vfield$$

We assume that the *effect* of 'seeing' an object is to make its relative distance and an angle *observable:*

$$\begin{aligned} obs(dist(obj)) &\stackrel{\text{def}}{=} see(obj) \\ obs(angle(obj)) &\stackrel{\text{def}}{=} see(obj) \end{aligned}$$

In addition, the agent always knows whether it's seeing an object or not:

$$obs(see(obj)) \stackrel{\text{def}}{=} \mathbf{true}$$

Finally we assume that the action *move* has the precondition

$$move \supset \neg blocked$$

[3] Note that *see* is modeled as a 'defined condition' rather than as a 'knowledge gathering action' as in [13]. In this way, the agent continuously gathers information from its surrounding (when certain conditions hold) without requiring its active participation in the form of deliberate actions.

where $blocked$ is defined by the collection of clauses:

$$blocked \textbf{ if } facing(obj) \wedge close(obj)$$

obtained by replacing obj by each identifier in OBJ, including the walls that are modeled as obstacles.

We are ready to prove that under some conditions, the program 2 will be *executable* and will *achieve* the goal condition $reached(goal)$. The conditions that we assume are:

$OBJ = \{goal\}$
$vfield = 360^0$
$dist(goal) \leq vdist$
$\delta angle < vfield/2$ **and** $\delta dist < vdist$**.**

We prove first that the action constraint $move \supset \neg blocked$ is satisfied in all states s_i of every model s_0, s_1, ...of the resulting control theory. Indeed, since there is only one object, the $goal$ object, $blocked$ is true iff $facing(goal)$ and $close(goal)$ are true. Yet that means that $reached(goal)$ is true, and then, $done$ is true and $move$ is false. Thus, $move \supset \neg blocked$ is always satisfied.

We prove now that the agent has always the *knowledge* to evaluate the conditions in the program. Let us prove first that the truth of $reached(goal)$ is always known. We consider two cases: when $see(goal)$ is true and when $see(goal)$ is false. In the second case, $reached(goal)$ must be false because the restrictions $\delta angle < vfield/2$ and $\delta dist < vdist$ guarantee that $\neg see(goal)$ implies $\neg close(goal) \vee \neg facing(goal)$. Likewise, in the first case, $dist(goal)$ and $angle(goal)$ must be known, and hence, the atoms $close(goal)$ and $facing(goal)$, as well as $reached(goal)$, must be known too. As a result, we get that if $see(goal)$ is false, $reached(goal)$ must be false, and if $see(goal)$ is true, $reached(goal)$ must be known. Since the state of the atom $see(goal)$ is always known, this means that the state of $reached(goal)$ will be known too. Similar arguments suffice to prove that the other conditions ($facing(goal)$ and $\neg blocked$) will also be known. The conclusion is that, given the initial conditions, the Program 7.1 is executable.

We are left to show that, under the assumptions above, the program will lead the agent to a state where $reached(goal)$ is true. Actually, under those assumptions $see(goal)$ must be initially true. Furthermore, since there is full angular visibility, and the agent only moves in the direction of the goal, once $see(goal)$ becomes true, it will remain true throughout. That means, among other things, that eventually the condition $facing(goal)$ will become true. Yet once this condition becomes true, the agent will move towards the goal, and every time the agent moves, $dist(goal)$ will decrease [4]. This does *not* mean, however, that once $facing(goal)$ becomes true it remains true until $goal$ is reached. Yet, even if $facing(goal)$ becomes false, the distance to the goal decreases and the program will bring back the agent to a state where $facing(goal)$ is true again by the effect of the action $rotate$. Thus, eventually, in a finite number of steps, $reached(goal)$ will be true.

---

[4] this involves some trigonometrical arguments based on the fact that the maximum $\delta angle$ must be 60^0 to assure that the move won't increase $dist(goal)$

7.2 Situation 2

If the assumption that the visibility range is 360^0 is changed and a limited range is used instead, the Program 2 remains executable but does not necessarily lead the agent to the goal. This is because, in the new context, it is no longer true that once $see(goal)$ is true, it remains true throughout. Indeed, if the agent is not facing the goal, it will rotate, and at certain point $see(goal)$ may become false. At that point, the agent will move away from the goal and won't come back (since we are assuming that there are no other objects such as walls).

Given this analysis, there is a simple modification to the program to deal with this problem. It consists in the introduction of two different rotations, a left rotation (counter-clockwise) and a right rotation (clockwise). Under the conditions above, even with a limited visibility range, the new program can be shown to lead the agent to the goal.

Program 3 specialization of the action *rotate*

$$\begin{aligned}
reached(goal) &\rightarrow done \\
see(goal) \wedge facing(goal) &\rightarrow move \\
see(goal) \wedge on\text{-}left(obj) &\rightarrow rotate\text{-}l \\
see(goal) \wedge on\text{-}right(obj) &\rightarrow rotate\text{-}r \\
\neg blocked &\rightarrow move \\
\mathbf{true} &\rightarrow rotate\text{-}l
\end{aligned}$$

The action rules for the two new actions are:

$$\begin{aligned}
rotate\text{-}r &\rightarrow \mathsf{angle} - \delta angle \\
rotate\text{-}l &\rightarrow \mathsf{angle} + \delta angle
\end{aligned}$$

and the definitions of the new conditions are:

$$\begin{aligned}
on\text{-}left(obj) &\stackrel{\text{def}}{=} 0 \leq angle(obj) \leq vfield/2 \\
on\text{-}right(obj) &\stackrel{\text{def}}{=} (2\pi - vfield/2) \leq angle(obj) \leq 2\pi
\end{aligned}$$

7.3 Situation 3

In the presence of other objects, i.e., when OBJ includes other objects besides *goal*, the situation changes significantly. First of all, the Program 2 is not executable, as in certain situations a *move* can be triggered in situations in which the precondition $\neg blocked$ is false. This however is easy to fix: we just need to introduce the condition $\neg blocked$ among the conditions for *move*.

The other problem (or feature!) is that the way that the model defines $see(obj)$ implicitly assumes that all objects are *transparent*; namely, the visibility of the agent (i.e., the definition of $see(goal)$) depends on the linear and angular distance to the goal; but do not take into account the presence of objects in the way. This implies that if $see(goal)$ is always true (say because the parameters $vfield$ and *vdist* are sufficiently large), the agent will *move* only when $facing(goal)$ and $\neg blocked$ are both true, and will *rotate* otherwise. However, when the obstacle in the way is perpendicular to the line that joins the agent to the goal, rotations alone cannot establish the conjunction $facing(goal) \wedge \neg blocked$ (because in such arrangement $facing(goal)$ will be true exactly when $facing(obst)$ is true, where $obst$ is the obstacle identifier). Thus in this case, the program will be make the robot *rotate* forever.

A solution to this problem can be obtained by replacing the line

$$see(goal) \rightarrow rotate$$

with the line

$$see(goal) \wedge \neg will\_block \rightarrow rotate$$

where $will\_block$, is defined as:[5]

$$will\_block \;\textbf{if}\; close(obj) \wedge angle(obj) \leq 2\delta angle$$

detects rotations that will make the condition *blocked* true, and in those cases, make the agent take a step away from the obstacle.

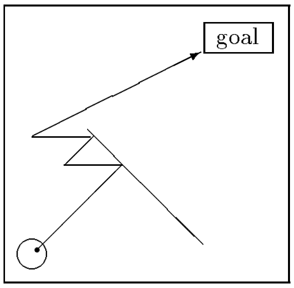

Fig. 2. 'Fly' behavior due to obstacles assumed 'transparent'

A typical 'run' of the resulting program is shown in Figure 2, where the agent exhibits the typical behavior of a fly trying to get passed a window.

The modeling of 'opaque' objects as opposed to 'transparent' objects requires a redefinition of $see(obj)$ to:

$$\begin{aligned} see(obj) \stackrel{\text{def}}{=} \; & dist(obj) \leq vdist \wedge \\ & angle(obj) \leq vfield(obj) \wedge \neg occluded(obj) \end{aligned}$$

where the *occluded* predicate must be defined in terms of the intersection between the visual region of the agent, the position of the object to *see* and the obstacles. In the presence of such obstacles, the agent can be trapped in trajectories in which the goal object is never visible, and thus, in which the program will not lead the robot to the goal. This can be seen in the snapshot taken from the simulator (Figure 3) built to experiment with the theories, models and programs posed in this work. The simulator is available in the WWW[6].

[5] For the condition $will\_block$ to be known, $2\delta angle$ should be smaller than $vfield$.

[6] The address is omitted in favor of the blind review process

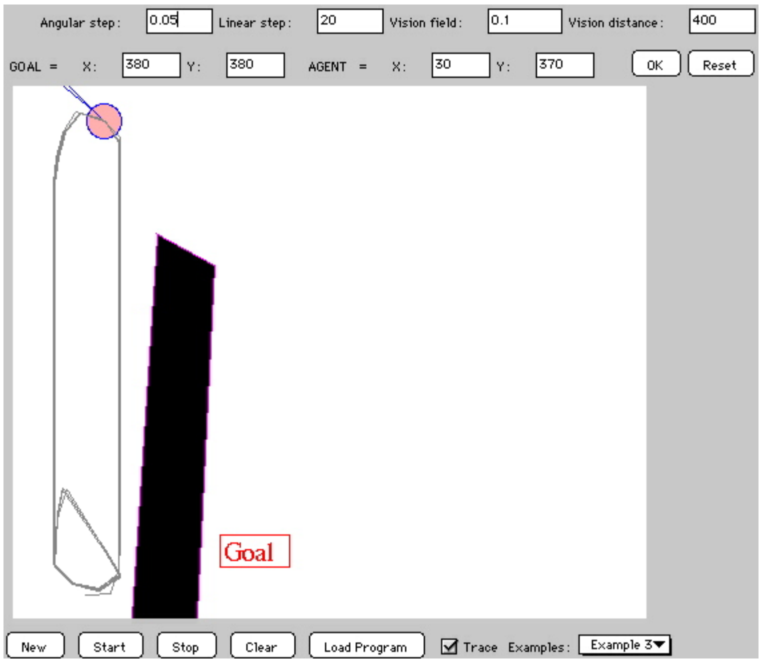

Fig. 3. 'Wall following' behavior due to opaque obstacles

8 Conclusions and Related Work

We have used a theory of actions and knowledge to analyze control programs for navigation tasks, modeling both physical and sensing actions and establishing the conditions under which different programs are executable and lead the agent to the goal. The use of action models can be useful also for the *construction* of programs. Nilsson's [12], for example, advocates a methodology for writing teleoreactive control programs in which the actions in one line are supposed to contribute to the realization of the conditions in the preceding lines (goal regression property [11, pp. 288–296]). This design criterion can be formalized in this framework as:

Definition 4. A program P is *teleoreactive* if in all models s_0, s_1, s_2, ..., every state s_i that makes $c_j \to a_j$ for $j > 0$ the first applicable condition-action pair in s_i, is followed after a finite number of time points by a state $s_{i+\Delta}$ that makes $c_k \to a_k$ the first applicable condition-action pair where $k < j$.

Our analysis shows clearly that, even in simple programs as Program 1, that satisfies the *universal* property of teleoreactive programs [12], there are conditions that violates the teleoreactive principle, since once the robot start moving, $facing(goal)$ may become temporarily false. However, these execution errors won't prevent (in this case) that the goal condition will ultimately be achieved.

For this type of models to be useful for the analysis of more realistic robot plans, richer control structures (e.g., see [9, 5, 2]) and uncertainty (e.g., [1, 14, 5]) will be needed. We plan to explore some of these issues, as well as the automatic construction of robot plans, elsewhere.

References

1. F. Bacchus, J. Halpern, and H. Levesque. Reasoning about noisy sensors in the situation calculus. In *Proceedings IJCAI-95*, pages 1933–1940, 1995.
2. C. Baral. Relating theories of action and reactive control. In *Proceedings AAAI-96 Workshop on Theories of action, Planning and Robot Control: Bridging the gap*, 1996.
3. R.A. Brooks. The behavior language; user's guide. Technical report, MIT AI Lab, 1990.
4. R.E. Fikes and N. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
5. H. Geffner and J. Wainer. A model of actions, knowledge and contingent plans. Technical report, Universidad Simón Bolívar, 1997.
6. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.
7. J.C. Latombe. *Robot Motion Planning*. KAP, 1991.
8. Y. Lesperance and H. Levesque. Indexical knowledge and robot action. *Artificial Intelligence*, 73:69–116, 1995.
9. H. Levesque. What is planning in the presence of sensing? In *Proceedings AAAI-96*, pages 1139–1146, 1996.
10. R. Moore. A formal theory of knowledge and action. In J. Hoobs and R. Moore, editors, *Formal Theories of the Commonsense World*. 1985.
11. N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., 1980.
12. N. Nilsson. Teleo-reactive programs for agent control. *JAIR*, 1:139–158, 1994.
13. R. Scherl and Levesque. H. The frame problem and knowledge producing actions. In *Proceedings AAAI-93*, pages 689–695, 1993.
14. M. Shanahan. Noise and the commonsense informatic situation for a mobile robot. In *Proceedings AAAI-96*, pages 1098–1103, 1996.
15. Y. Shoham. Agent oriented programming. Technical report, Robotics Lab, Computer Science Department, Stanford University, 1993.
16. S. Thrun. To know or not to know on the utility of models in mobile robots. *AI Magazine*, Spring:47–54, 1997.