On Deliberation under Incomplete Information and the Inadequacy of Entailment and Consistency-Based Formalizations *

Giuseppe De Giacomo
Dip. Informatica e Sistemistica
School of Computer Science and Engineering
Univer. di Roma "La Sapienza"
Via Salaria 113, 00198 Roma
Italy
degiacomo@dis.uniroma1.it

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

Hector J. Levesque
Sebastian Sardiña
Dept. of Computer Science
University of Toronto
Toronto, ON, M5S 3G4
Canada
hector@cs.toronto.edu
ssardina@cs.toronto.edu

ABSTRACT

Much of the work in agent programming assumes an execution model where an agent has a knowledge base (KB) about the current state of the world, and makes decisions about what to do in terms of what is entailed or consistent with this KB. Deliberation or planning then would involve looking ahead and gauging what would be consistent or entailed at various stages by future KBs. We show that in the presence of sensing, this account of deliberation does not work properly, and propose an alternative that does.

Keywords

Agent programming, deliberation, semantics, situation calculus

1. INTRODUCTION

There has been considerable work on formal models of deliberation/planning under incomplete information, where the agent can perform sensing actions to acquire additional information. This problem is very important in agent applications such as web information retrieval/management. However, much of the previous work on formal models of deliberation — models of knowing how, ability, epistemic feasibility, etc. such as [16, 3, 10, 14, 5] — has been set in epistemic logic-based frameworks and is hard to relate to work on agent programming languages (e.g. 3APL [9], AgentSpeak(L) [19], etc.). In this paper, we develop new non-epistemic formalizations of deliberation that are much closer and easier to relate to standard agent programming language semantics based on transition systems. Our results show that the commonly held view that deliberation can simply be taken as a different control regime involving search over the agent program's transition tree is fundamentally flawed in the presence of sensing actions.

When doing deliberation/planning under incomplete information, one typically searches over a set of states, each of which is associated with a knowledge base (KB) or theory that represents what is known in the state. To evaluate tests in the program and to determine what transitions/actions are possible, one looks at what is *entailed* by the KB. To allow for sensing results, one looks at which of these are *consistent* with the KB. We call this type of approach to deliberation "entailment and consistency-based" (EC-based). In this paper, we show that EC-based approaches do not work in general, and propose an alternative. Our accounts are formalized within the situation calculus and use ConGolog to specify agent programs as described in Section 2, but we claim that the results generalize to most proposed agent programming languages/frameworks. We point out that this paper is mainly concerned with the semantics of the deliberation process and not much with the actual algorithms implementing such process.

In Sections 3 and 4, we develop a simple EC-based account of deliberation ($KHow_{EC}$). We show that this account gives the wrong results on a simple example involving indefinite iteration. Then, we show that whenever this account says that a deliberation/planning problem is solvable, there is a *conditional plan* (a finite tree program without loops) that is a solution. It follows that this account is limited to problems where the total number of steps needed can be bounded in advance. We claim that this limitation is not specific to the simple account and applies to all EC-based accounts of deliberation.

The source of the problem with the EC-based account is the use of local consistency checks to determine which sensing results are possible. This does not correctly distinguish between the models that satisfy the overall domain speci-

*(Produces the permission block, copyright information and page numbering). For use with ACM_PROC_ARTICLE-SP.CLS V2.6SP. Supported by ACM.

fication (for which the plan must work) and those that do not. To get a correct account of deliberation, one must take into account what is true in different models of the domain together with what is true in all of them (what is entailed). In Section 5, we develop such an entailment and truth-based account ($KHow_{ET}$), argue that it intuitively does the right thing, and show how it correctly handles our test examples. Following this, we return to discuss the meaning of these results for agent programming language semantics.

2. THE SITUATION CALCULUS AND INDIGOLOG

The technical machinery we use to define program execution in the presence of sensing is based on that of [6, 4]. The starting point in the definition is the situation calculus [15]. 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. We assume that actions return binary sensing results, and we use the predicate $SF(a, s)$ to characterize what the action tells the agent about its environment. For example, the axiom

$$Near(d, s) \supset [SF(senseDoor(d), s) \equiv Open(d, s)]$$

states that if the agent is near a door d then the action $senseDoor(d)$ tells it whether the door is open. For actions with no useful sensing information, we write $SF(a, s) \equiv True$.

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 $Poss(a, s)$.
- Guarded successor state axioms of the form $\alpha(\vec{x}, a, s) \supset [F(\vec{x}, do(a, s)) \equiv \gamma(\vec{x}, a, s)]$ providing the usual solution to the frame problem but only when the guard holds.
- Guarded sensed fluent axioms of the form $\alpha(\vec{x}) \supset [SF(a, s) \equiv \phi(\vec{x}, s)]$ as described above.
- Unique names axioms for the primitive actions.
- A set of foundational, domain independent axioms for situations Σ as in [20].

Such theories are essentially a special case of the guarded action theories of [7], which generalize the basic action theories of [20] to allow for fluents that change in unpredictable ways under certain conditions and deal with sensing.[1]

To describe a run of a program which includes both actions and their sensing results, we use the notion of a *history*, i.e., a sequence of pairs (a, x) where a is a primitive action and x is 1 or 0, a sensing result. Intuitively, the history $\sigma = (a_1, x_1) \cdot \ldots \cdot (a_n, x_n)$ is one where actions $a_1, \ldots, a_n$ happen starting in some initial situation, and each action a_i returns sensing value x_i. We use $end[\sigma]$ for the situation term defined by: $end[\epsilon] = S_0$; and $end[\sigma \cdot (a, x)] = do(a, end[\sigma])$. We use $Sensed[\sigma]$ for the formula of the situation calculus defined by: $Sensed[\epsilon] = True$; and $Sensed[\sigma \cdot (a, 1)] = Sensed[\sigma] \wedge SF(a, end[\sigma])$, and $Sensed[\sigma \cdot (a, 0)] = Sensed[\sigma] \wedge \neg SF(a, end[\sigma])$.

Next we turn to programs. The programs we consider here are based on the ConGolog language defined in [4], an extension of Golog [12] providing a rich set of programming constructs. 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 [2], a character specification program for computer animation [8], a softbot application in the personal banking domain [11], and others. In this paper, we will only need the following procedure-free subset:

α,	primitive action
$\phi?$,	wait for a condition
$\delta_1; \delta_2$,	sequence
if ϕ **then** δ_1 **else** δ_2 **endIf**,	conditional
while ϕ **do** δ **endWhile**,	while loop
$\delta_1 \mid \delta_2$,	nondeterministic branch
$\pi\, x.\, \delta$,	nondeterministic choice of argument
δ^*,	nondeterministic iteration
$\delta_1 \parallel \delta_2$,	concurrency

In [4], a single step transition semantics in the style of [18] 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 [13, 4], are characterized using the $Do(\delta, s, s')$ predicate, which means that there is an execution of program δ 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 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 [4].

[1] Guarded theories, however, provide a passive account of sensing by using online sensors instead of specific sensing actions.

Observe that an offline executor has no access to sensing results, available only at runtime. IndiGolog, an extension of ConGolog to deal with online executions with sensing, is proposed in [6]. The semantics defines an *online execution* of a program δ starting from a history σ, as a sequence of *online configurations* $(\delta_0 = \delta, \sigma_0 = \sigma), \ldots, (\delta_n, \sigma_n)$ such that for $i = 0, \ldots, n-1$:

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma_i]\} \models Trans(\delta_i, end[\sigma_i], \delta_{i+1}, end[\sigma_{i+1}]),$$

$$\sigma_{i+1} = \begin{cases} \sigma_i & \text{if} \quad end[\sigma_{i+1}] = end[\sigma_i], \\ \sigma_i \cdot (a, x) & \text{if} \quad end[\sigma_{i+1}] = do(a, end[\sigma_i]) \\ & \quad \text{and } a \text{ returns } x. \end{cases}$$

An *online execution successfully terminates* if

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma_n]\} \models Final(\delta_n, end[\sigma_n]).$$

In Golog and ConGolog, a programmer can write a nondeterministic program and rely on the interpreter to do lookahead to find a way to successfully execute the program; this can be used for planning. In IndiGolog, lookahead is not automatic. Instead, a *search* operator $\Sigma(\delta)$ is introduced to allow the programmer to specify when lookahead should be performed. We do not present the formal details here. However, suffice it to say that this operator only allows a transition for δ if there exists a sequence of further transitions that would allow δ to terminate successfully. Unfortunately, this fails to rule out cases where a sequence must exist but where the agent cannot determine what the sequence is. A more satisfactory notion of deliberation is presented in [5], and is the basis for the ideas presented here.

3. DELIBERATION: EC-BASED ACCOUNT

Perhaps the first approach to come to mind for defining when an agent knows how/is able to execute a program δ in a history σ goes as follows: the agent must be able to repeatedly choose some action that is known to be executable and allowed by the program, such that no matter what sensing output is obtained as a result of doing the action, he can continue this process with what remains of the program and eventually reach a configuration where he knows he can legally terminate. We can formalize this idea as follows.

We define $KHow_{EC}(\delta, \sigma)$ to be the smallest relation $\mathcal{R}(\delta, \sigma)$ such that:

(A) for all pairs (δ, σ), if

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Final(\delta, end[\sigma])$$

then $\mathcal{R}(\delta, \sigma)$;

(B) for all pairs (δ, σ), if there exists δ' such that

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Trans(\delta, end[\sigma], \delta', end[\sigma])$$

and $\mathcal{R}(\delta', \sigma)$, then $\mathcal{R}(\delta, \sigma)$;

(C) for all pairs (δ, σ), if there exist δ' and an action a such that

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Trans(\delta, end[\sigma], \delta', do(a, end[\sigma]))$$

and for all sensing results μ such that $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma \cdot (a, \mu)]\}$ is consistent, it is the case that $\mathcal{R}(\delta', \sigma \cdot (a, \mu))$, then $\mathcal{R}(\delta, \sigma)$.

Note that here, the agent's lack of complete knowledge in a history σ is modeled by the theory $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\}$ being incomplete and having many different models. $KHow_{EC}$ uses entailment to ensure that the agent *knows what* transition he may perform next. For instance, for a conditional program involving different actions a_1 and a_2 in the "then" and "else" branches (i.e., such that $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models a_1 \neq a_2$), the agent must know whether the test holds and know how to execute the appropriate branch:

$$\begin{array}{l} KHow_{EC}(\textbf{if } \phi \textbf{ then } a_1 \textbf{ else } a_2 \textbf{ endIf}, \sigma) \text{ iff} \\ \quad \mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models \phi(end[\sigma]) \text{ and} KHow_{EC}(a_1, \sigma) \\ \qquad\qquad\qquad\qquad \text{or} \\ \quad \mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models \neg\phi(end[\sigma]) \text{ and} KHow_{EC}(a_2, \sigma). \end{array}$$

$KHow_{EC}$ uses consistency to determine which sensing results can occur, for which the agent needs to have a subplan that leads to a $Final$ configuration. Due to this, we say that $KHow_{EC}$ is an *entailment and consistency-based* (EC-based) account of knowing how.

We can easily define a notion of when an agent *can achieve a goal* ϕ *in a history* σ, $Can_{EC}(\phi, end[\sigma])$, based on this as follows:

$$Can_{EC}(\phi, \sigma) \text{ iff } KHow_{EC}(\textbf{while } \neg\phi \textbf{ do } (\pi a.a) \textbf{ end}, \sigma),$$

i.e., the agent knows how to execute the program that involves repeatedly choosing and executing some action until the goal has been achieved. Note how this shows that ability to achieve a goal is a special case of knowing how to execute a (nondeterministic) program.

The EC-based account of deliberation seems quite intuitive and attractive. However it has a fundamental limitation: it fails on programs involving indefinite iteration. The following simple example taken from [10] shows the problem.

Consider a situation in which an agent wants to cut down a tree. Assume that the agent has a primitive action *chop* to chop at the tree, and also assume that he can always find out whether the tree is down by doing the (binary) sensing action *look*. If the sensing comes 1, then the tree is down; otherwise the tree still remains up. There is also a fluent $RemainingChops(s)$, which we assume ranges over the natural numbers $\mathbb{N}$ and is meant to represent how many *chop* actions are still required in s to bring the tree down. The agent's goal is to bring the tree down, which means bringing about a situation s such that the $Down(s)$ holds. The abbreviation $Down(s)$ is an defined as follows:

$$Down(s) \stackrel{\text{def}}{=} RemainingChops(s) = 0$$

The action theory $\mathcal{D}_{tc}$ is the union of:

1. The foundational axioms for situations Σ.
2. $\mathcal{D}_{una} = \{chop \neq look\}$.

3. $\mathcal{D}_{ss}$ contains the following successor state axiom:

$$RemainingChops(do(a,s)) = n \equiv \\ a = chop \wedge RemainingChops(s) = n+1 \vee \\ a \neq chop \wedge RemainingChops(s) = n.$$

4. $\mathcal{D}_{ap}$ contains the following two precondition axioms:

$$Poss(chop,s) \equiv True, \\ Poss(look,s) \equiv (RemainingChops > 0).$$

5. $\mathcal{D}_{S_0} = \{RemainingChops(S_0) \neq 0\}$.

6. $\mathcal{D}_{sf}$ contains the following two sensing axioms:

$$SF(chop,s) \equiv True, \\ SF(look,s) \equiv (RemainingChops(s) = 0).$$

Notice that sentence $\exists n.RemainingChop(S_0) = n$ (where the variable n ranges over $\mathbb{N}$) is entailed by this theory so "infinitely" hard tree trunks are ruled out. Nonetheless, the theory *does not* entail the sentence $RemainingChop(S_0) < k$ for any constant $k \in \mathbb{N}$. Hence, there exists some $n \in \mathbb{N}$, though unknown and unbounded, such that the tree will fall after n chops. Because of this, intuitively, we should have that the agent can bring the tree down $Can_{EC}(Down, S_0)$, since if the agent keeps chopping, the tree will eventually come down, and the agent can find out whether it has come down by looking. Moreover, for the program

$$\delta_{tc} = \textbf{while } \neg Down \textbf{ do } chop; look \textbf{ endWhile}$$

we should have that $KHow_{EC}(\delta_{tc}, \epsilon)$ (note that δ_{tc} is deterministic). However, this is not the case:

THEOREM 3.1. *Let δ_{tc} be the above mentioned program to bring the tree down. Then, for all $k \in \mathbb{N}$, $KHow_{EC}(\delta_{tc}, ((chop,1) \cdot (look,0))^k)$ does not hold. In particular, when $k = 0$, $KHow_{EC}(\delta_{tc}, \epsilon)$ does not hold.*

Thus, the simple EC-based formalization of knowing how gives the wrong result for this example. Why is this so? Intuitively, it is easy to check that if $KHow_{EC}(\delta_{tc}, \epsilon)$, then for all $j \in \mathbb{N}$, $KHow_{EC}(\delta_{tc}, ((chop,1) \cdot (look,0))^j)$ and $KHow_{EC}(look; \delta_{tc}, ((chop,1) \cdot (look,0))^j \cdot (chop,1))$. Now consider a possible execution where an agent keeps chopping and looking and seeing that the tree is not down forever. There is no model of $\mathcal{D}_{tc}$ where δ_{tc} yields this execution, as the tree is guaranteed to come down after a finite number of chops. However, by the above, we see that $KHow_{EC}$ is in fact taking this execution into account in determining whether the agent knows how to execute δ_{tc}, since every finite prefix of the execution is indeed consistent with $\mathcal{D}_{tc}$. The problem is that the set of *all* of them together is not. This is why $KHow_{EC}$ fails. In the next section, we show that $KHow_{EC}$'s failure on the tree chopping example is due to a general limitation of the $KHow_{EC}$ formalization. Note that Moore's original account of ability [16] is closely related to $KHow_{EC}$ and also fails on the tree chopping example [10].

4. $KHow_{EC}$ CAN ONLY HANDLE BOUNDED SOLUTIONS

In this section, we show that whenever $KHow_{EC}(\delta, \sigma)$ holds for some program δ and history σ, there is simple kind of conditional plan, what we call a *TREE* program, that can be followed to execute δ in σ. Since for *TREE* programs (and conditional plans), the number of steps they perform can be bounded in advance (there are no loops), it follows that $KHow_{EC}$ will never be satisfied for programs whose execution cannot be bounded in advance. Since there are many such programs (for instance, the one for the tree chopping example), it follows that $KHow_{EC}$ is fundamentally limited as a formalization of knowing how and can only be used in contexts where attention can be restricted to bounded strategies. As in [5], we define the class of *(sense-branch) tree programs TREE* with the following BNF rule:

$$dpt ::= nil \mid False? \mid a; dpt_1 \mid True?; dpt_1 \mid \\ sense_\phi; \textbf{if } \phi \textbf{ then } dpt_1 \textbf{ else } dpt_2$$

where a is any non-sensing action, and dpt_1 and dpt_2 are tree programs.

This class includes conditional programs where one can only test a condition that has just been sensed (or trivial tests, which are introduced only for technical reasons). Thus as shown in [5], whenever a *TREE* program is executable, it is also epistemically feasible, i.e., the agent can execute it without ever getting stuck not knowing what transition to perform next. As well, *TREE* programs are clearly deterministic.

Let us define a relation $KHowBy_{EC} : Program \times History \times TREE$. The relation is intended to associate a program δ and history σ for which $KHow_{EC}$ holds with some *TREE* program(s) that can be used as a strategy for successfully executing δ in σ.

We define $KHowBy_{EC}(\delta, \sigma, \delta^{tp})$ to be the least relation $\mathcal{R}(\delta, \sigma, \delta^{tp})$ such that:

(a) If $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Final(\delta, end[\sigma])$, then $\mathcal{R}(\delta, \sigma, nil)$.

(b) If $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Trans(\delta, end[\sigma], \delta', end[\sigma])$ and $\mathcal{R}(\delta', \sigma, \delta^{tp\prime})$, then $\mathcal{R}(\delta, \sigma, True?; \delta^{tp\prime})$.

(c) If $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Trans(\delta, end[\sigma], \delta', do(a, end[\sigma]))$ and for all μ such that $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma \cdot (a,\mu)]\}$ is consistent, there exists δ^{tp}_μ such that $\mathcal{R}(\delta', \sigma \cdot (a,\mu), \delta^{tp}_\mu)$ then $\mathcal{R}(\delta, \sigma, a; \textbf{if } \phi \textbf{ then } \delta^{tp\prime} \textbf{ else } \delta^{tp\prime} \textbf{ endIf})$ where ϕ is the condition on the right hand side of the sensed fluent axiom for a, and if $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma \cdot (a,1)]\}$ is inconsistent then $\delta^{tp\prime} = False?$ else $\delta^{tp\prime} = \delta^{tp}_1$ and if $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma \cdot (a,0)]\}$ is inconsistent then $\delta^{tp\prime} = False?$ else $\delta^{tp\prime} = \delta^{tp}_0$.

We first show that for every program δ and history σ for which $KHow_{EC}(\delta, \sigma)$ holds, there is a program δ^{tp} such that $KHowBy_{EC}(\delta, \sigma, \delta^{tp})$ holds (we show later that δ^{tp} must be a *TREE* program):

THEOREM 4.1. *For all programs δ and histories σ, if $KHow_{EC}(\delta,\sigma)$, then there exists a program δ^{tp} such that $KHowBy_{EC}(\delta,\sigma,\delta^{tp})$.*

Next, we show that whenever $KHowBy_{EC}(\delta,\sigma,\delta^{tp})$ holds, then δ^{tp} is a *TREE* program which is guaranteed to terminate in a *Final* situation of the given program δ (in all models), and $KHow_{EC}(\delta,\sigma)$ holds. Note that *TREE* programs are by definition deterministic, and thus this means that all executions of δ^{tp} must be executions of δ as well (which execution we get depends only on the sensing results obtained).

THEOREM 4.2. *For all programs δ, histories σ, and programs δ^{tp}, if $KHowBy_{EC}(\delta,\sigma,\delta^{tp})$ then we have that*

- *δ^{tp} is a TREE program,*
- *$KHow_{EC}(\delta,\sigma)$,*
- $\mathcal{D}\cup\mathcal{C}\cup\{Sensed[\sigma]\}\models \exists s.Do(\delta^{tp},end[\sigma],s)\wedge Do(\delta,end[\sigma],s)$.

Since the number of steps a *TREE* program performs can be bounded in advance, it follows that $KHow_{EC}$ will never hold for programs/problems that are solvable, but whose execution requires a number of steps that cannot be bounded in advance, for instance, the program in the tree chopping example. Thus $KHow_{EC}$ is severely restricted as an account of knowing how; it can only be complete when all possible strategies are bounded.

5. DELIBERATION: ET-BASED ACCOUNT

We saw in Section 3 that the reason $KHow_{EC}$ failed on the tree chopping example was that it required the agent to have a choice of action that guaranteed reaching a *Final* configuration even for histories such as σ_ω that were inconsistent with the domain specification. There was a branch in the configuration tree that corresponded to that that history. This occurred because "local consistency" was used to construct the configuration tree. The consistency check kept switching which model of $\mathcal{D}\cup\mathcal{C}$ (which may be thought as representing the environment) was used to generate the next sensing result, postponing the observation that the tree had come down forever. But in the real world, sensing results come from a fixed environment (even if we don't know which environment this is). It seems reasonable that we could correct the problem by fixing the model of $\mathcal{D}\cup\mathcal{C}$ used in generating possible configurations in our formalization of knowing how. This is what we will now do.

Assume for the time being that we are only dealing with deterministic programs, i.e. programs that do not use the nondeterministic constructs $|$, π, *, and $\|$. We define when an agent knows how to execute a program δ in a history σ and model M (which represents the environment), $KHowInM(\delta,\sigma,M)$, as the smallest relation $\mathcal{R}(\delta,\sigma)$ such that:

(A) for all pairs (δ,σ), if
$$\mathcal{D}\cup\mathcal{C}\cup\{Sensed[\sigma]\}\models Final(\delta,end[\sigma])$$
then $\mathcal{R}(\delta,\sigma)$;

(B) for all pairs (δ,σ), if there exists δ' such that
$$\mathcal{D}\cup\mathcal{C}\cup\{Sensed[\sigma]\}\models Trans(\delta,end[\sigma],\delta',end[\sigma])$$
and $\mathcal{R}(\delta',\sigma)$, then $\mathcal{R}(\delta,\sigma)$;

(C) for all pairs (δ,σ), if there exist δ' and an action a such that
$$\mathcal{D}\cup\mathcal{C}\cup\{Sensed[\sigma]\}\models Trans(\delta,end[\sigma],\delta',do(a,end[\sigma]))$$
and if $M\models SF(a,end[\sigma])$ then it is the case that $\mathcal{R}(\delta',\sigma\cdot(a,1))$ and if $M\models\neg SF(a,end[\sigma])$ then it is the case that $\mathcal{R}(\delta',\sigma\cdot(a,0))$, then $\mathcal{R}(\delta,\sigma)$.

The only difference between this and $KHow_{EC}$ is that the sensing results come from the fixed model M. Given this, we obtain the following formalization of when an agent knows how to execute a program δ in a history σ:

$KHow_{ETd}(\delta,\sigma)$ iff for every model M such that $M\models\mathcal{D}\cup\mathcal{C}\cup\{Sensed[\sigma]\}$, $KHowInM(\delta,\sigma,M)$.

We call this type of formalization *entailment and truth-based,* since it uses entailment to ensure that the the agent knows what transitions he can do, and *truth in a model* to obtain possible sensing results.

$KHow_{ETd}$ actually works for programs δ that are deterministic. For instance, it handles the tree chopping example correctly: $KHow_{ETd}(\delta_{tc},\epsilon)$ holds. However, for nondeterministic programs, $KHow_{ETd}$ is too weak. Consider the following example. There is a treasure behind one of two doors but the agent does not know which. We want to know if the agent knows how to execute the program δ_{treas}:

$$[(open1;look)\mid(open2;look)];AtTreasure?$$

Intuitively, the agent does not know how to execute δ_{treas} because he does not know which door to open to get to the treasure. However, $KHow_{ETd}(\delta_{treas},\epsilon)$ holds. Indeed in a model M_1 where the treasure is behind door 1, the agent can pick the *open*1 action, and then we have $KHowInM(look;AtTreasure?,[(open1,1)],M_1)$, and thus $KHowInM(\delta_{treas},\epsilon,M_1)$. Similarly, in a model M_2 where the treasure is behind door 2, he can pick *open*2, and thus $KHowInM(\delta_{treas},\epsilon,M_2)$.

The problem with $KHow_{ETd}$ for nondeterministic programs is that the action chosen need not be the same in different models even if they have generated the same sensing results up to that point and are indistinguishable for the agent.[2] We can solve this problem by requiring that the agent have a common strategy for all models/environments, i.e., that

[2]Note that $KHow_{EC}$ does not suffer from this problem and works just fine for nondeterministic programs, provided they can only execute a bounded number of steps.

he have a *deterministic* program δ^d that he knows how to execute (in all models of the theory) and knows will terminate in a $Final$ situation of the given program δ:

$KHow_{ET}(\delta, \sigma)$ iff there is a deterministic program δ^d such that $KHow_{ETd}(\delta^d, \sigma)$ and

$$\begin{aligned}&\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models \\ &\qquad \exists s. Do(\delta^d, end[\sigma], s) \wedge Do(\delta, end[\sigma], s).\end{aligned}$$

We do not think that it is possible to obtain a much simpler general formalization of knowing how and to avoid the quantification over deterministic programs/strategies. A notion of ability to achieve a goal can be defined in terms of the $KHow_{ET}$ account as we did for $KHow_{EC}$ in Section 3.

6. DISCUSSION

What are the implications of our results for work on agent programming languages (e.g. 3APL [9], AgentSpeak(L) [19], etc.)? The semantics of such languages are usually specified as a transition system. For instance in 3APL, configurations are pairs involving a program and a belief base, and a transition relation over such pairs is defined by a set of rules. Evaluating program tests is done by checking whether they are entailed by the belief base. Checking action preconditions is done by querying the agent's belief base update relation, which would typically involve determining entailments over the belief base — the 3APL semantics abstracts over the details of this. Sensing is not dealt with explicitly, although one can suppose that it could be handled by simply updating the belief base (AgentSpeak(L) has events for this kind of thing).

Most work in this area only deals with on-line reactive execution, where no deliberation/lookahead is performed; this type of execution just involves repeatedly selecting some transition allowed in the current configuration. However, a commonly held view is that *deliberation can simply be taken as a different control regime involving search over the agent program's transition tree.* It is understood that in the presence of sensing, one needs to find more than just a path to a final configuration in the transition tree; one needs some sort of plan/subtree where the agent has chosen some transition among those allowed but must have branches for all possible sensing results. The natural way of determining which sensing results are possible is checking their consistency with the current belief base. This is essentially an EC-based approach. Also in work on planning under incomplete information, e.g. [1, 17], a similar sort of setting is typically used, and finding a plan involves searching a (finite) space of knowledge states that are compatible with the planner's knowledge. Whereas the planner in [1] uses BDDs to represent the set of possible states, the planner in [17] uses restricted formulas instead. In any case, the underlying models are meant to represent only the *current* possible states of the environment, which, in turn, are updated upon the hypothetical execution of an action at planning time. This is quite different from the way models are used here. We use models that are dynamic in the sense that they represent the potential responses of the environment for *any* future state. In that way, then, what the above planners are doing is deliberation in the style of $KHow_{EC}$. Our results show that this view of deliberation is fundamentally flawed when sensing is present. It produces an account that only handles problems that can be solved in a bounded number of actions. As an approach to implementing deliberation, this may be perfectly fine. But as a semantics or specification, it is wrong. What is required is a much different kind of account, like our ET-based one.

We believe that there is a close relationship between $KHow_{ET}$ some of the earlier epistemic accounts of knowing how and ability [16, 3, 10, 14, 5]. We hope to get some correspondence results on this soon. The work presented here is also of great relevance for defining a more adequate search/deliberation operator in IndiGolog.

7. REFERENCES

[1] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of IJCAI-01*, pages 473–478, 2001.

[2] W. Burgard, A. B. Cremers, D. Fox, D. Hahnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, Madison, Wisconsin, 1998.

[3] E. Davis. Knowledge preconditions for plans. *Journal of Logic and Computation*, 4(5):721–766, 1994.

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

[5] G. De Giacomo, Y. Lespérance, H. J. Levesque, and S. Sardiña. On the semantics of deliberation in IndiGolog: From theory to implementation. In D. Fensel, F. Giunchiglia, D. McGuiness, and M.-A. Williams, editors, *Principles of Knowledge Representation and Reasoning, Proc. of the 8th Int. Conf. (KR2002)*, pages 603–614, Toulouse, France, April 2002. Morgan Kaufmann. Also appeared in Proc. AIPS'02.

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

[7] G. De Giacomo and H. J. Levesque. Progression and regression using sensors. In *Proc. of IJCAI-99*, pages 160–165, 1999.

[8] J. Funge. *Making Them Behave: Cognitive Models for Computer Animation*. PhD thesis, University of Toronto, Toronto, Canada, 1998.

[9] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. J. C. Meyer. A formal semantics for an abstract agent programming language. In *Proceedings of ATAL-97*, pages 215–229, 1998.

[10] Y. Lespérance, H. J. Levesque, F. Lin, and R. B. Scherl. Ability and knowing how in the situation calculus. *Studia Logica*, 66(1):165–186, 2000.

[11] Y. Lespérance, H. J. Levesque, and S. Ruman. An experiment in using golog to build a personal banking assistant. In L. Rao and W. Wobcke, editors, *Intelligent Agent Systems: Theoretical and Practical Issues*, volume 1209 of *Lectures Notes in Artificial Intelligence (LNAI)*, pages 27–43. Springer-Verlag, 1997.

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

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

[14] F. Lin and H. J. Levesque. What robots can do: Robot programs and effective achievability. *Artificial Intelligence*, 101:201–226, 1998.

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

[16] R. C. Moore. A formal theory of knowledge and action. In J. R. Hobbs and R. C. Moore, editors, *Formal Theories of the Common Sense World*, pages 319–358. Ablex Publishing, Norwood, NJ, 1985.

[17] R. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, pages 212–221, 2002.

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

[19] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logica computable language. In W. V. Velde and J. W. Perram, editors, *Agents Breaking Away (LNAI)*, volume 1038, pages 42–55. Springer-Verlag, 1996.

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