Ability and Knowing How in the Situation Calculus*

Yves Lespérance, Hector J. Levesque,†
Fangzhen Lin, and Richard B. Scherl‡

Department of Computer Science
University of Toronto
Toronto, ON, M5S 1A4 Canada
{lesperan,hector,fl,scherl}@ai.toronto.edu

January 1995

Abstract

Most agents can acquire information about their environments as they operate. A good plan for such an agent is one that not only achieves the goal, but is also executable, i.e., ensures that the agent has enough information at every step to know what to do next. In this paper, we presents a formal account of what it means for an agent to *know how to execute a plan* and to be *able to achieve a goal*. Such a theory is a prerequisite for producing specifications of planners for agents that can acquire information at run time. It is also essential to account for cooperation among agents. Our account is more general than previous proposals, handles "while loops" properly, and incorporates an approach to the frame problem. It can also be used to prove programs containing sensing actions correct.

*This research received financial support from the Information Technology Research Center (Ontario, Canada), the Institute for Robotics and Intelligent Systems (Canada), and the Natural Science and Engineering Research Council (Canada).

†Fellow of the Canadian Institute for Advanced Research.

‡Current address: Department of Computer and Information Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102 USA

1 Introduction

Work in the classical planning paradigm has generally made very strong assumptions about the domain in which planning is taking place, in particular, that the planner has complete knowledge of the initial state, and that actions are such that the planner can compute a complete description of any state reachable by doing a sequence of actions in the initial state (for instance, STRIPS operators). Such assumptions cannot be sustained in most real applications (e.g., robotics, information gathering agents); there, agents need to acquire knowledge at execution time by sensing their environment.

Some recent work, for instance [3], has attempted to generalize classical planning techniques to deal with this. But a key problem is that in such domains, it is not even clear what a plan is and when it is a solution to a particular planning problem. Plans must at the very least include conditional control structures so that the choice of action can depend on the result of sensing. But then it appears that standard programming language notions of correctness are insufficient. Even if it can be shown that a plan must achieve the goal (and terminate), the agent may not have enough knowledge to execute it. For example, suppose that the agent knows that behind one of two doors there is a treasure and behind the other there is a monster, but does not know which. Then even though the plan

$$\textbf{if } \textsc{TreasureBehindDoor1} \textbf{ then } \textsc{goThrough}(\textsc{Door1}) \textbf{ else } \textsc{goThrough}(\textsc{Door2})$$

can be shown to achieve the goal of getting the treasure, the agent does not *know how* to execute it because he cannot evaluate the test. Similarly,

$$\textsc{goThrough}(\textsc{DoorToTreasure})$$

achieves the goal, but cannot be executed because the agent does not know which primitive action the program stands for. The nondeterministic plan[1]

$$[\textsc{goThrough}(\textsc{Door1}) \mid \textsc{goThrough}(\textsc{Door2})]; \textsc{AtTreasure}?$$

also achieves the goal, but cannot be executed as the agent does not know which branch to take. However, if he can look through a window on one of the doors to determine what is behind it, then the following plan is adequate:

Example 1

$$\begin{array}{l} \textsc{lookThroughWindow}; \\ \textbf{if } \textsc{TreasureBehindDoor1} \textbf{ then } \textsc{goThrough}(\textsc{Door1}) \\ \qquad \textbf{else } \textsc{goThrough}(\textsc{Door2}) \end{array}$$

---

[1] Our use of test actions may be confusing to some; read $\phi?;\delta$ as "action δ occurring when ϕ holds," and for $\delta;\phi?$, read "action δ occurs after which ϕ holds." Thus, the plan in the example involves either going through DOOR1 or going through DOOR2, so that one ends up at the treasure.

It must achieve the goal and the agent will know how to execute it.

Whether an agent knows how to execute a plan depends on how smart he is — how much he knows and what sort of inferences he can perform. A very smart agent that can do lookahead would know how to execute the following nondeterministic plan:

Example 2

$$\text{LOOKTHROUGHWINDOW};$$
$$\textbf{pick } d : [\text{DOOR}(d)?; \text{GOTHROUGH}(d)];$$
$$\text{ATTREASURE}?$$

A dumber executor would not.

All this is really part of our common sense knowledge about agents. We do not delegate a goal to someone unless we believe he is *able* to achieve it. And even if someone does not know how to achieve a goal on his own, we may still enlist his help by providing instructions he knows how to follow. Such instructions would typically not specify the plan down to the last detail; we assume some intelligence on the part of the executor.

The classical planning paradigm involves a very smart planner and a very dumb executor — it is assumed that the difficult problem solving is performed at planning time, and that execution is relatively direct. But there is no real reason to restrict our attention to this picture. In some cases, planning from scratch may be so hard that it is better to try to build a smart executor that the user can program at a high level — we pursue this in [6]. Others have suggested that the right role for plans is as advice to a relatively smart improvisation module [1]. Also, multi-agent systems are becoming more common and typically involve agents at different levels of smartness. All this suggests studying what knowing how or ability means for agents with varying levels of intelligence.

Before one even starts talking about plans, it is useful to have a formal account of what sort of knowledge is involved in the ability to achieve a goal. This is what we develop in section 3. Plans are partial representations of this kind of knowledge; how complete they must be depends on how smart the intended executor is. In section 4, we develop two accounts of knowing how to execute a plan, one for a very smart agent and another for a much dumber one. In fact, these are merely two points in a space of agents with various kinds of abilities. But as argued in the concluding section, the framework we propose provides a useful foundation for further exploration of this space.

We will discuss related work as it becomes relevant. It is worth singling out, however, the very similar work of Ernest Davis [2]. Like us, Davis develops accounts of knowing how to execute a plan for both smart and dumb executors. However in [2], he fails to show that his account really handles unbounded iteration, a key problem area in earlier work. Nor does he discuss ability to achieve a goal and its relation to knowing how. Although developed independently, our accounts of knowing how are remarkably similar, and it seems that most of our results could have been obtained using his axiomatization as a starting point. We point out some of the differences as they become pertinent.

2 A Theory of Action

Our theory is based on an extended version of the situation calculus [8], a predicate calculus dialect for representing dynamically changing worlds. In this formalism, the world is taken to be in a certain state (or situation). That state can only change as a result of an agent doing an action. The term $do(a, s)$ represents the state that results from the agent's performance of action a in state s. For example, the formula $\mathrm{ON}(\mathrm{A}, \mathrm{B}, do(\mathrm{PUTON}(\mathrm{A}, \mathrm{B}), s))$ could mean that A is on B in the state that results from the agent's doing $\mathrm{PUTON}(\mathrm{A}, \mathrm{B})$ in state s. Predicates and function symbols whose value may change from state to state (and whose last argument is a state) are called *fluents*.

An action is specified by first stating the conditions under which it can be performed by means of a *precondition axiom*. For example,

$$Poss(\mathrm{PICKUP}(x), s) \equiv \forall z \neg \mathrm{HOLDING}(z, s) \wedge \mathrm{NEXTTO}(x, s)$$

means that it is possible for the agent to pick up an object x in state s iff he is not holding anything and is standing next to x in s. Then, one specifies how the action affects the world's state with *effects axioms*, for example:

$$Poss(\mathrm{DROP}(x), s) \wedge \mathrm{FRAGILE}(x) \supset \mathrm{BROKEN}(x, do(\mathrm{DROP}(x), s)).$$

Note that we write $s < s'$ iff s' is the result of doing some sequence of actions in s, where the actions are possible in the situation where they are done.

The above axioms are not sufficient if one wants to reason about change. It is usually necessary to add frame axioms that specify when fluents remain unchanged by actions. The frame problem [8] arises because the number of these frame axioms is of the order of the product of the number of fluents and the number of actions. Our approach incorporates a treatment of the frame problem due to Reiter [12] (who extends previous proposals by Pednault [11], Schubert [14] and Haas [5]). The basic idea behind this is to collect all effects axioms about a given fluent and assume that they specify all the ways the value of the fluent may change. A syntactic transformation can then be used to obtain a *successor state axiom* for the fluent, for example:

$$\begin{aligned} &Poss(a, s) \supset [\mathrm{BROKEN}(x, do(a, s)) \equiv \\ &\quad a = \mathrm{DROP}(x) \wedge \mathrm{FRAGILE}(x) \vee \mathrm{BROKEN}(x, s) \wedge a \neq \mathrm{REPAIR}(x)]. \end{aligned}$$

This says that x is broken after the agent does action a in state s iff either the action was dropping x and x is fragile, or x was already broken in s and the action was not repairing it. This treatment avoids the proliferation of axioms, as it only requires a single successor state axiom per fluent and a single precondition axiom per action.[2]

Scherl and Levesque [13] have generalized this account to handle knowledge-producing actions. Such actions affect the mental state of the agent rather than the state of the external world. For example, after performing the action SENSEDOWN, an agent would know whether the tree he is trying to cut is down (**KWhether**(ϕ, s) stands for **Know**$(\phi, s) \vee$ **Know**$(\neg\phi, s)$):

$$Poss(\text{SENSEDOWN}, s) \supset \textbf{KWhether}(\text{DOWN}, do(\text{SENSEDOWN}, s)).$$

Similarly, after doing READCOMBOFSAFE an agent might know what the combination of the safe he is trying to open is:

$$\begin{array}{l} Poss(\text{READCOMBOFSAFE}, s) \supset \\ \quad \exists c\, \textbf{Know}(\text{COMBOFSAFE} = c, do(\text{READCOMBOFSAFE}, s)). \end{array}$$

Knowledge is represented by adapting the possible world model to the situation calculus (as first done by Moore [9]). $K(s', s)$ represents the fact that in state s, the agent thinks the state of the world could be s'. **Know**(ϕ, s) is an abbreviation for the formula $\forall s'(K(s', s) \supset \phi(s'))$. For clarity, we sometimes use the pseudo-variable now to represent the state bound by the enclosing **Know**; so **Know**$(\text{DOWN}(\mathsf{now}), s)$ stands for $\forall s'(K(s', s) \supset \text{DOWN}(s'))$. We require K to be transitive and euclidean, which ensures that the agent always knows whether he knows something (i.e., positive and negative introspection).

For a domain with the two sensing actions described above, the successor state axiom for the knowledge fluent K can be specified as follows:

$$\begin{array}{l} Poss(a, s) \supset (K(s^*, do(a, s)) \equiv \\ \quad \exists s'[K(s', s) \wedge s^* = do(a, s') \wedge Poss(a, s') \wedge \\ \qquad (a = \text{SENSEDOWN} \supset (\text{DOWN}(s') \equiv \text{DOWN}(s))) \wedge \\ \qquad (a = \text{READCOMBOFSAFE} \supset \text{COMBOFSAFE}(s') = \text{COMBOFSAFE}(s))]). \end{array}$$

First note that for non-knowledge-producing actions (e.g. DROP(x)), the specification ensures that the only change in knowledge that occurs in moving from s to $do(\text{DROP}(x), s)$ is the knowledge that the action DROP has been successfully performed. For the case of a knowledge-producing action such as SENSEDOWN, the idea is that in moving from s to $do(\text{SENSEDOWN}, s)$, the agent not only knows that the action has been performed (as above), but also the truth value of the associated predicate DOWN. Since in this case we require that $\text{DOWN}(s') \equiv \text{DOWN}(s)$, DOWN will have the same truth value in all s' such that $K(do(\text{SENSEDOWN}, s'), do(\text{SENSEDOWN}, s))$. Observe that for any situation s, DOWN is true at $do(\text{SENSEDOWN}, s)$ iff DOWN is true at s. Therefore, DOWN has the same truth value in all worlds s^* such that

[2] This discussion ignores the ramification problem; a treatment compatible with our approach has been proposed by Lin and Reiter [7].

$K(s^*, do(\text{SENSEDOWN}, s))$, and so **KWhether**$(\text{DOWN}, do(\text{SENSEDOWN}, s))$ holds. Similar reasoning explains why we must have $\exists c\,$**Know**$(\text{COMBOFSAFE} = c, do(\text{READCOMBOFSAFE}, s))$. This can be extended to an arbitrary number of knowledge-producing actions in a straightforward way.

3 Ability

Very roughly, ability to achieve a goal involves knowing what to do when, so as to arrive at a goal state. We make this more precise by appealing to the notion of an *action selection function,* a mapping from situations to primitive actions. We understand such a function as prescribing which action the agent should perform in a situation. We say that situation s' is on the path prescribed by action selection function σ in situation s iff there is a path from s to s' and at every step along the way, the action performed is the one prescribed by σ:

$$\textbf{OnPath}(\sigma, s, s') \stackrel{\text{def}}{=} s \leq s' \wedge \forall a \forall s^*(s < do(a, s^*) \leq s' \supset \sigma(s^*) = a).$$

Note that **OnPath**(σ, s, s') implies that all the actions prescribed by σ between s and s' are possible.

We will say that the agent "can get" to a state where a goal ϕ holds by following action selection function σ in state s iff there is a situation s' on the path prescribed by σ in s where the agent knows that the goal holds, and at every step between s and s', the agent knows what the next action prescribed by σ is:

$$\begin{aligned}\textbf{CanGet}(\phi, \sigma, s) \stackrel{\text{def}}{=} \exists s'(&\textbf{OnPath}(\sigma, s, s') \wedge \textbf{Know}(\phi, s') \wedge \\ &\forall s^*[s \leq s^* < s' \supset \exists a\, \textbf{Know}(\sigma(\mathsf{now}) = a, s^*)]).\end{aligned}$$

Finally, we say that the agent can achieve a goal ϕ in situation s iff there exists an action selection function σ such that he knows in s that he can get to a state where the goal holds by following σ:

$$\textbf{Can}(\phi, s) \stackrel{\text{def}}{=} \exists \sigma\, \textbf{Know}(\textbf{CanGet}(\phi, \sigma, \mathsf{now}), s).$$

For the example sketched in the introduction, where an agent wants to get to a treasure but does not know which of two doors leads to it, it is straightforward to verify that our definition yields the right results, i.e., that the agent can achieve the goal iff it is possible for him to sense whether the treasure is behind a given door. Our account also gives the right results for more challenging examples involving unbounded iteration, such as the following:

Example 3 Consider a situation where an agent wants to cut down a tree. We assume that the tree will fall down after some unknown number of primitive chopping actions. This yields the following definition and axioms:

$$\text{DOWN}(s) \stackrel{\text{def}}{=} \text{REMAININGCHOPS}(s) = 0,$$

$$Poss(a, s) \supset [\text{REMAININGCHOPS}(do(a, s)) = n \equiv$$
$$a = \text{CHOP} \wedge \text{REMAININGCHOPS}(s) = n + 1 \vee$$
$$a \neq \text{CHOP} \wedge \text{REMAININGCHOPS}(s) = n],$$

$$Poss(\text{CHOP}, s) \equiv \text{REMAININGCHOPS}(s) > 0.$$

We also assume that the agent can always find out whether the tree is down by sensing. This yields the following successor state axiom for K and precondition axiom for SENSEDOWN:

$$Poss(a, s) \supset (K(s^*, do(a, s)) \equiv$$
$$\exists s'[K(s', s) \wedge s^* = do(a, s') \wedge Poss(a, s') \wedge$$
$$(a = \text{SENSEDOWN} \supset (\text{DOWN}(s') \equiv \text{DOWN}(s)))]),$$

$$Poss(\text{SENSEDOWN}, s) \equiv True.$$

Notice however that we do not assume that the agent knows how many chop actions are necessary to get the tree down. Even then, it seems that the agent should be able to achieve the goal of cutting the tree down; all he needs to do is to keep sensing and chopping until the tree is down. Indeed, it is straightforward to verify that the above axioms imply that **Can**(DOWN, S_0): Consider the action selection function such that $\sigma(s)$ is CHOP whenever $\exists s^*\, s = do(\text{SENSEDOWN}, s^*)$, and SENSEDOWN otherwise. It is easy to show that the agent must always know what action is prescribed by σ. And since in any belief alternative REMAININGCHOPS chops are sufficient to get the tree down, it follows that the agent can get to a goal state by following σ. ■

Example 4 Now suppose that the agent has no way of sensing whether the tree is down. Then we get the following successor state axiom for K:

$$Poss(a, s) \supset (K(s^*, do(a, s)) \equiv$$
$$\exists s'[K(s', s) \wedge s^* = do(a, s') \wedge Poss(a, s')]).$$

Suppose also that $\neg$**Know**(DOWN, S_0). Then we would expect the agent to be unable to get the tree down. Indeed, it can be verified that $\neg$**Can**(DOWN, S_0): the assumptions imply that **Know**($\forall s^*(\mathsf{now} \leq s^* \supset \neg$**Know**(DOWN, s^*)), S_0); by the definition of **Can**, the result follows. ■

To our knowledge, this is the first time an account has been shown to handle both ability and inability in cases involving unbounded iteration. The earlier accounts of Moore [9] and Morgenstern [10] have problems with such cases; we explain their inadequacies in the next section.

Let us now examine some properties of our definition of ability and see how some alternative definitions fail to handle important cases. To simplify the discussion, for the remainder of this section we will be assuming that all actions are possible, i.e., $\forall a \forall s Poss(a, s)$. Our results could easily be generalized. If one were to try to give an inductive definition of **Can**, one would likely start from the observations that:

- if a goal is known to hold already, then it can be achieved, and
- if there is an action such that the agent knows that he can achieve the goal after the action is performed, then he can achieve the goal from the beginning.

In fact, we have shown that given our definition, **Can** holds iff one of the above conditions hold:

Proposition 5

$$\mathbf{Can}(\phi, s) \equiv (\mathbf{Know}(\phi, s) \vee \exists a\, \mathbf{Know}(\mathbf{Can}(\phi, do(a, \mathsf{now})), s)).$$

Note that establishing this result (in either direction) requires the assumption that agents have negative introspection (i.e., that K is euclidean). This is one point over which our account differs from Davis's [2], so the proposition would not hold in his system.

The above result might suggest a simpler way of defining ability: use the above equivalence as an axiom to somehow define **Can**. Unfortunately, this approach does not seem to work. By itself, the axiom is too weak; for instance, it is consistent with it that **Can** (for any given goal) is always true. If on the other hand, we try to define ability as the least fixed-point of the above equivalence, the resulting version of ability ends up being too strong. Let

$$\begin{aligned}&\mathbf{Can}_{\perp}(\phi, s) \stackrel{\text{def}}{=} \forall C(\\ &\quad \forall s'[C(s') \equiv \mathbf{Know}(\phi, s') \vee \exists a\, \mathbf{Know}(C(do(a, \mathsf{now})), s')] \supset C(s)).\end{aligned} \tag{1}$$

Now using proposition 5, it is easy to show that $\mathbf{Can}_{\perp}$ is stronger than **Can**: $\forall s(\mathbf{Can}_{\perp}(\phi, s) \supset \mathbf{Can}(\phi, s))$. However, $\mathbf{Can}_{\perp}$ is not implied by **Can**. In fact, $\mathbf{Can}_{\perp}$ fails to handle our tree chopping example — we get that $\neg\mathbf{Can}_{\perp}(\text{DOWN}, S_0)$ despite the fact that intuitively, the agent can get the tree down by repeatedly sensing and chopping. To see this, take C to be true of a situation iff the tree is known to be down in that situation. Then C clearly satisfies the equivalence in (1). But this means that $\mathbf{Can}_{\perp}$ will be true in no additional situations, as it is a least fixed point. Since the tree is not down in the initial state S_0, this means that $\mathbf{Can}_{\perp}$ is false in S_0.

Historically, our definition of **Can** was motivated by $\mathbf{Can}_{\perp}$, and its failure on the tree example. It remains an open question whether there is a natural fixed-point equation like the equivalence inside (1) for which **Can** is the least fixed point solution. We also considered an iterative analogue to $\mathbf{Can}_{\perp}$; it too failed to handle the tree example properly.

4 Knowing How

To get help from other agents in achieving our goals, we often need to give them explicit instructions, some sort of program to execute. Whether an agent knows how to execute a program depends on how smart the agent is. We will now formalize some notions of knowing how that appear significant; towards the end, we also relate knowing how to ability to achieve a goal.

4.1 Programs in the Extended Situation Calculus

Our programs will include the following nondeterministic forms:

$\delta_1 \vert \delta_2$	nondeterministic choice of branch
$\pi x\, \delta(x)$	nondeterministic choice of argument
$\pi a\, \delta(a)$	nondeterministic choice of primitive action

To be able to talk about the different deterministic execution paths through a nondeterministic program, we will extend our earlier notion of action selection function. Let a *path selection function* σ be a mapping from situations into pairs of objects and actions. To simplify our notation, for any path selection function σ, and any situation s, we denote the left member of $\sigma(s)$ as $\sigma_l(s)$, and the right member as $\sigma_r(s)$, i.e. $\sigma(s) = (\sigma_l(s), \sigma_r(s))$. We will use σ_l to pick an object in interpreting $\pi x\, \delta(x)$ and similarly for σ_r and $\pi a\, \delta(a)$. To handle $\delta_1|\delta_2$, we introduce a reserved action constant symbol $null$; we will take the left branch iff $\sigma_r(s) = null$. Semantically, $null$ behaves like a no-op, and has no effects.

We introduce programs into the formalism as abbreviations (macros). The abbreviation $Do(\delta, \sigma, s, s')$, where δ is a program and σ is a path selection function, means that the execution of δ according to σ starting in state s terminates in the state s'. It is defined inductively as follows:

$$Do(\theta, \sigma, s, s') \stackrel{\text{def}}{=} Poss(\theta, s) \wedge s' = do(\theta, s), \text{ provided } \theta \text{ is an action term.}$$

$$Do(\phi?, \sigma, s, s') \stackrel{\text{def}}{=} \phi(s) \wedge s' = s.$$

$$Do(\delta_1; \delta_2, \sigma, s, s') \stackrel{\text{def}}{=} \exists s''(Do(\delta_1, \sigma, s, s'') \wedge Do(\delta_2, \sigma, s'', s')).$$

$$Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, \sigma, s, s') \stackrel{\text{def}}{=}$$
$$(\phi(s) \supset Do(\delta_1, \sigma, s, s')) \wedge (\neg\phi(s) \supset Do(\delta_, \sigma, s, s'))$$

$$Do(\delta_1|\delta_2, \sigma, s, s') \stackrel{\text{def}}{=} (\sigma_r(s) = null \ \supset Do(\delta_1, \sigma^+, s, s')) \wedge$$
$$(\sigma_r(s) \neq null \ \supset Do(\delta_2, \sigma^+, s, s')).$$

$$Do(\pi x\, \delta(x), \sigma, s, s') \stackrel{\text{def}}{=} Do(\delta(\sigma_l(s)), \sigma^+, s, s').$$

$$Do(\pi a\, \delta(a), \sigma, s, s') \stackrel{\text{def}}{=} Do(\delta(\sigma_r(s)), \sigma^+, s, s').$$

$$Do(\textbf{while } \phi \textbf{ do } \delta, \sigma, s, s') \stackrel{\text{def}}{=}$$
$$\forall P\{\forall s_1(\neg\phi(s_1) \supset P(s_1, s_1)) \wedge$$
$$\forall s_1, s_2, s_3(\phi(s_1) \wedge Do(\delta, \sigma, s_1, s_2) \wedge P(s_2, s_3) \supset P(s_1, s_3))\} \supset P(s, s').$$

Here σ^+ is defined by the following axiom:

$$\forall s\, \sigma^+(s) = \sigma(do(null, s)).$$

This is needed in order to properly handle cases like $(A|B)|C$ and $\pi x(\pi y\, A(x, y))$. So the $null$ action plays two roles: it handles the nesting of $|$ and π operators by advancing the path selection function after each selection, and as a possible value of a path selection function, it is used to select which branch of $\delta_1|\delta_2$ one should take.

Given a complex action δ and a path selection function σ, there is at most one terminating state:

Proposition 6 $Do(\delta, \sigma, s, s_1) \wedge Do(\delta, \sigma, s, s_2) \supset s_1 = s_2$.

If $|$ and π do not occur in a program δ, we say that it is *determinate*. It is clear from the definition that path selection functions play no role in the interpretation of determinate programs:

Proposition 7

$$\text{If } \delta \text{ is determinate, then } \forall \sigma, \sigma', s, s'(Do(\delta, \sigma, s, s') \equiv Do(\delta, \sigma', s, s')).$$

We write $Do(\delta, s, s')$ when executing δ in s leads to s' under some path selection function: $Do(\delta, s, s') \stackrel{\text{def}}{=} \exists \sigma\, Do(\delta, \sigma, s, s')$. Then from the proposition we have that

$$\text{if } \delta \text{ is determinate, then } \forall \sigma, s, s'(Do(\delta, s, s') \equiv Do(\delta, \sigma, s, s')).$$

In formalizing knowing how, we must consider not just terminating states, but also all intermediate states. We shall use the abbreviation $During(\delta, \sigma, s, s')$ to mean that state s' occurs during the execution of δ starting in s according to σ. If there is a state s^* such that $Do(\delta, \sigma, s, s^*)$ holds, then $During(\delta, \sigma, s, s')$ holds iff $s \leq s^* \leq s'$. However, we also want $During$ to hold for the states encountered in executions that do not successfully terminate. For non-terminating executions, all states encountered are $During$; so for example, $During($**while** $True$ **do** $null, \sigma, s, s')$ holds iff s' is a successor of s where only $null$ actions happen between s and s'. For executions that terminate unsuccessfully, all states between the starting state and the one where the program fails are $During$; for example, $During(\text{STACKONTO}(\mathrm{A}, \mathrm{B}); False?, \sigma, s, s')$ holds iff s' is s or $do(\text{STACKONTO}(\mathrm{A}, \mathrm{B}), s)$. We omit the formal definition, which is very similar to that given for Do.

4.2 Executability under a Strategy

A path selection function specifies a kind of execution strategy. We say that an agent *can execute* a program when he follows *a given strategy* iff the program terminates when executed according to the strategy and at every point during the execution, either the agent knows the program has terminated or knows which action to take next. We define this formally as follows:[3]

[3]Another way of understanding this is the following: the combination of a nondeterministic program and an execution strategy stands for the deterministic specialization of the program obtained by executing it with the strategy; then **CanExec**(δ, σ, s) stands for ability to execute the deterministic program referred to by $\langle \delta, \sigma \rangle$.

$$\begin{aligned}&\textbf{CanExec}(\delta, \sigma, s) \stackrel{\text{def}}{=} \exists s^* \, Do(\delta, \sigma, s, s^*) \wedge \\ &\quad \forall s_i(During(\delta, \sigma, s, s_i) \supset \\ &\qquad \{\forall s', s'_i[K(s', s) \wedge K(s'_i, s_i) \wedge s' \leq s'_i \supset Do(\delta, \sigma, s', s'_i)] \vee \\ &\qquad\; \exists a \forall s', s'_i[K(s', s) \wedge K(s'_i, s_i) \wedge s' \leq s'_i \supset During(\delta, \sigma, s', do(a, s'_i))]\}).\end{aligned}$$

Note that an agent may be able to execute a program according to a strategy without knowing in advance that the program will terminate:

$$\not\models \textbf{CanExec}(\delta, \sigma, s) \supset \textbf{Know}(\exists s' \, Do(\delta, \sigma, \mathsf{now}, s'), s).$$

For example, consider the program SENSE_P; **while** $\neg P$ **do** $null$. Assume that P holds initially but the agent is not aware of that, i.e., $P(S_0) \wedge \neg\textbf{Know}(P, S_0)$. Then the agent can execute the program in S_0 because after doing SENSE_P, he will know that P holds, and will not enter the infinite while loop. But initially, the agent does not know that the program will terminate, because as far as he is concerned, it may well be the case that $\neg P$. This implies that an agent may be able to execute a program according to a strategy without realizing that this is the case:

$$\not\models \textbf{CanExec}(\delta, \sigma, s) \supset \textbf{Know}(\textbf{CanExec}(\delta, \sigma, now), s).$$

It is also worth noting that since the execution of determinate programs does not depend on the execution strategy, we have:

Proposition 8

For all determinate programs δ, $\exists\sigma\textbf{CanExec}(\delta, \sigma, s) \supset \forall\sigma\textbf{CanExec}(\delta, \sigma, s)$.

4.3 Dumb Knowing How

One way an agent may execute a possibly nondeterministic program is by arbitrarily picking an alternative at every choice point. Since we cannot rule out any execution strategy, we must require that he be able to execute the program according to *all* strategies to ensure he will succeed. This ability to blindly execute a program is what we call *dumb knowing how*. We define the notion formally as follows:

$$\textbf{DKH}(\delta, s) \stackrel{\text{def}}{=} \forall\sigma[\forall s' \exists x \, \textbf{Know}(\sigma(\mathsf{now}) = x, s') \supset \textbf{CanExec}(\delta, \sigma, s)].$$

Note that we only consider path selection functions whose value is always known to the agent, that is, strategies that the agent knows how to follow. With respect to the situation described earlier where someone is seeking a treasure, a dumb agent knows how to execute the program in example 1 but not the one in example 2.

One can show that if an agent can blindly execute a program, then the program must terminate no matter what execution strategy is used:

Proposition 9 $\forall s(\mathbf{DKH}(\delta, s) \supset \forall\sigma\exists s'\, Do(\delta, \sigma, s, s'))$.

The **DKH** notion is particularly useful for cases where an agent wants to delegate a task to another agent. For instance, in a cooperative environment, agent A may come up with a plan to achieve one of his goals, make sure that agent B knows how to dumbly execute the plan, and then asks B to do it. If B executes the program, he will eventually terminate and be able to go on to other business and A's goal will be achieved. (B, having faith in agent A, need not know that he knows how to execute the program; he can simply trust agent A on this.) A special case is when A and B are the same agent. Then the agent knows that he knows how to dumbly execute the program, i.e., $\mathbf{Know}(\mathbf{DKH}(\delta, \mathsf{now}), s)$.

4.4 Smart Knowing How

Another way an agent may execute a possibly nondeterministic program is by considering ahead of time whether there are alternatives at every choice point whose choice guarantees that he will be able to execute the program. Such an ideal agent is looking ahead before committing to any execution strategy. It seems that if such an agent knows of some strategy that he can execute the program under the strategy, then we can be confident that he will pick that strategy (or some equally good one) and succeed in executing the program. We call this ability to smartly execute a program *smart knowing how*. It is defined formally as follows:

$$\mathbf{SKH}(\delta, s) \stackrel{\text{def}}{=} \exists\sigma\, \mathbf{Know}(\mathbf{CanExec}(\delta, \sigma, \mathsf{now}), s).$$

For instance, a smart agent does know how to execute the program in example 2 (as well as that in example 1 of course). However, no agent will ever know how to execute $False?|$**while** $True$ **do** $null$, because neither of its branches can be executed; the first one fails and the second one loops forever.

An immediate consequence of the definition is that if an agent knows how to smartly execute δ, then he knows that δ has a terminating state:

Proposition 10 $\mathbf{SKH}(\delta, s) \supset \mathbf{Know}(\exists s'\, Do(\delta, \mathsf{now}, s'), s)$.

We mentioned earlier that the accounts proposed by Moore [9] and Morgenstern [10] are inadequate for dealing with unbounded iteration. The problem arises with non-terminating programs such as **while** $True$ **do** a. Intuitively, we would want to say that no agent knows how to execute such a program, as it is impossible to bring it to termination. Our account conforms to this and yields $\neg\mathbf{SKH}(\mathbf{while}\ True\ \mathbf{do}\ a, s)$ as well as $\neg\mathbf{DKH}(\mathbf{while}\ True\ \mathbf{do}\ a, s)$. The axioms provided by Moore and Morgenstern however, do not rule out an agent's knowing how to execute such a program.

4.5 Relationships among these Notions

It is interesting to examine the relationships among these notions. First, if one knows that one knows how to dumbly execute a program, then one knows how to execute it in a smart way:

Proposition 11

$$\text{for all complex actions } \delta,\ \mathbf{Know}(\mathbf{DKH}(\delta, \mathsf{now}), s) \supset \mathbf{SKH}(\delta, s)$$

The converse does not hold in general because there may be strategies under which the program cannot be executed and a smart executor will be able to avoid these, while a dumb one will not. However, since the execution of determinate programs is independent of any strategy, we have:

Proposition 12

$$\text{for all determinate complex actions } \delta,\ \mathbf{Know}(\mathbf{DKH}(\delta, \mathsf{now}), s) \equiv \mathbf{SKH}(\delta, s)$$

Perhaps the most interesting feature of our account is that smart knowing how can be related in a very natural way to the notion of ability to achieve a goal defined earlier. Let us define

$$Achieve(\phi) \stackrel{\text{def}}{=} \textbf{while } \neg\mathbf{Know}(\phi) \textbf{ do } \pi a\, a.$$

$Achieve(\phi)$ is a kind of universal program for achieving the goal ϕ. Then, we can show that being able to achieve a goal is equivalent to knowing how to achieve it by executing the universal program:

Proposition 13 $\mathbf{Can}(\phi, s) \equiv \mathbf{SKH}(Achieve(\phi), s)$.

This is an appealing property. We could take this as a definition for **Can**, but we find our earlier definition simpler and easier to work with.

5 Planning reconsidered

In this paper, we presented a definition of ability and two definitions of knowing how as macro abbreviations in the situation calculus, and showed that they had reasonable formal properties and generalized a number of other accounts. These definitions, we claimed, were a necessary first step to any theory of planning in a context involving incomplete knowledge of the initial state, knowledge-producing actions, and actions with context-dependent effects.

What our account does not provide, however, is a theory of planning itself. What exactly is a plan? If we simply say that it is any program that achieves a goal and that the agent knows how to execute, then for smart agents, the planning problem is absolutely trivial: when $\neg\mathbf{Can}(\phi, s)$, there can be no plan for ϕ; but when **Can**(ϕ, s), the agent also knows how to execute the universal program defined above: **SKH**$(Achieve(\phi), s)$.

For dumber agents, however, the case is not so clear. Even if **Can**(ϕ, s), there is no guarantee that there even exists a program δ that will bring about ϕ and such that **DKH**(δ, s) holds. What would be ideal in this case would be a way of synthesizing a suitable program from a proof of **Can**(ϕ, s), that is, from a proof of $\exists\sigma$ **Know**(**CanGet**$(\phi, \sigma, \mathsf{now}), s)$. This is can be thought of as a generalization of answer extraction [4] that would somehow convert a selection function into a program of the appropriate sort.

We can also imagine a variety of types of programs for agents of varying power. For a very dumb agent, we might require that all tests in all if-then-elses and while-loops in the program consist of comparisons among known sensor values. This would decouple the agent from any background theory of the world. Another alternative might be to allow tests that refer to values of fluents, and assume that the agent can use successor state axioms at run time. Yet another possibility is to allow tests and actions that incorporate limited versions of planning. For instance, we might let the agent decide at run time whether or not it needs to perform a knowledge-producing action before executing a test. There is clearly a tradeoff here: the more we assume of our agent at execution time (with whatever effects on performance this might have), the less work will be necessary at planning time.

References

[1] Philip E. Agre and David Chapman. What are plans for? *Robotics and Autonomous Systems*, 6:17–34, 1990.

[2] Ernest Davis. Knowledge preconditions for plans. Technical Report 637, Computer Science Department, New York University, 1993.

[3] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 115–125, Cambridge, MA, 1992. Morgan Kaufmann Publishing.

[4] C.C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 183–205. American Elsevier, New York, 1969.

[5] Andrew R. Haas. The case for domain-specific frame axioms. In F.M. Brown, editor, *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, pages 343–348, Lawrence, KA, April 1987. Morgan Kaufmann Publishing.

[6] Yves Lespérance, Hector J. Levesque, F. Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl. A logical approach to high-level robot programming – a progress report. In Benjamin Kuipers, editor, *Control of the Physical World by Intelligent Agents, Papers from the 1994 AAAI Fall Symposium*, pages 79–85, New Orleans, LA, November 1994.

[7] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994.

[8] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, UK, 1979.

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

[10] Leora Morgenstern. Knowledge preconditions for actions and plans. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 867–874, Milan, Italy, August 1987. Morgan Kaufmann Publishing.

[11] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In R.J. Brachman, H.J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, Toronto, ON, May 1989. Morgan Kaufmann Publishing.

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

[13] Richard B. Scherl and Hector J. Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 689–695, Washington, DC, July 1993. AAAI Press/The MIT Press.

[14] L.K. Schubert. Monotonic solution to the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In H.E. Kyberg, R.P. Loui, and G.N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, Boston, MA, 1990.

A Additional Proofs and Results

A.1 Proof of Proposition 5

The proof uses three lemmas. Remember that for simplicity here, we are assuming that primitive action are always physically possible. First, we show that whenever a goal is known to already hold, it can be achieved:

Lemma 14 $\models \forall s(\textbf{Know}(\phi, s) \supset \textbf{Can}(\phi, s))$.

Proof: This follows from the fact that since $\textbf{Know}(\phi, s)$ holds, executing $Achieve(\phi)$ in any state s' that is accessible from s, i.e. $K(s', s)$ terminates immediately without ever entering the while loop. ■

Then, we show that if there is an action such that the agent knows that he can achieve his goal after the action is performed, then he can achieve the goal from the beginning:

Lemma 15

$$\models \forall s(\exists a\, \textbf{Know}(\textbf{Can}(\phi, do(a, \textsf{now})), s) \supset \textbf{Can}(\phi, s)).$$

Proof: Suppose that $\textbf{Know}(\textbf{Can}(\phi, do(a, now)), s)$. Then we have that

$$\forall s'(K(s', s) \supset \textbf{Can}(\phi, do(a, s'))),$$

so that

$$\forall s'(K(s', s) \supset \textbf{SKH}(Achieve(\phi), do(a, s'))),$$

thus

$$\forall s'(K(s', s) \supset \exists \sigma_{s'}\, \textbf{Know}(\textbf{CanExec}(Achieve(\phi), \sigma_{s'}, now), do(a, s'))) \quad (*)$$

i.e., for every K-accessible state s', there is a action selection function $\sigma_{s'}$ that the agent knows will get her to the goal. We will show that $\textbf{Can}(\phi, s)$, by constructing a single action selection function that works for every accessible state.

First, notice that we can partition the accessible states into equivalence classes according to whether they remain mutually accessible after the performance of action a. Given s_1 and s_2 such that $K(s_1, s)$ and $K(s_2, s)$, let $s_1 \approx s_2$ iff $K(do(a, s_2), do(a, s_1))$. It is straightforward to show that $\approx$ must be an equivalence relation given the successor state axiom for K and the requirement that K be transitive and euclidean. We must select a single action selection function for all states in a given equivalence class in order to construct a global action selection function according to which the agent can execute $Acheiev(\phi)$. Let f be some arbitrary function that maps an equivalence class into the action selection function associated with one of its member, i.e., such that $f([s_1]) = \sigma_{s_2}$ where $s_1 \approx s_2$. We claim that $\forall s'(K(s', s) \supset \textbf{CanExec}(Achieve(\phi), f([s']), do(a, s')))$, i.e., in every accessible state the agent can execute $Achieve(\phi)$ according to the action selection function selected by f after doing a. To see this, suppose that $f([s']) = \sigma_{s*}$; then $K(s^*, s)$ and $K(do(a, s'), do(a, s^*))$; so by $(*)$ **CanExec**$(Achieve(\phi), \sigma_{s*}, do(a, s'))$.

Now let us define a global action selection function as follows:

$$\sigma_r(s^*) = \begin{cases} f([s'])_r(s^*) & \text{if } \exists s'(K(s', s) \wedge s' < s^*) \\ a & \text{otherwise} \end{cases}$$

It follows that $\forall s'(K(s', s) \supset$ **CanExec**$(Achieve(\phi), \sigma, do(a, s')))$. Since $\forall s'(K(s', s) \supset \sigma_r(s') = a)$, we must also have that $\forall s'(K(s', s) \supset$ **CanExec**$(Achieve(\phi), \sigma, s'))$, and thus that **Can**$(\phi, s)$. ■

Finally, we show the converse of the above two results:

Lemma 16

$$\models \forall s(\textbf{Can}(\phi, s) \supset \textbf{Know}(\phi, s) \vee \exists a\, \textbf{Know}(\textbf{Can}(\phi, do(a, \textsf{now})), s)$$

Proof: We assume that **Can**(ϕ, s) and $\neg$**Know**(ϕ, s) and show that

$$\exists a\, \textbf{Know}(\textbf{Can}(\phi, do(a, now)), s).$$

>From the first assumption, we have

$$\exists \sigma\, \textbf{Know}(\textbf{CanExec}(Achieve(\phi), \sigma, now), s). \quad (*)$$

Suppose s^* is a state such that $K(s^*, s)$. Such a state must exist because K is euclidean. Let $\sigma_r(s^*) = a$. Because $\neg$**Know**(ϕ, s), $\neg Do(Achieve(\phi), \sigma, s^*, s^*)$. Thus from (*), we have that

$$(\forall s')K(s', s) \supset (\sigma_r(s') = a \wedge During(Achieve(\phi), \sigma, s', do(a, s'))).$$

Thus from (*) and the definition of the program $Achieve(\phi)$, we have that

$$(\forall s')K(s', s) \supset \textbf{CanExec}(Achieve(\phi), \sigma, do(a, s')).$$

By our assumption about K, we then have that

$$(\forall s')K(s', s) \supset (\forall s'')(K(s'', do(a, s')) \supset \textbf{CanExec}(Achieve(\phi), \sigma, s'')).$$

Thus

$$(\forall s')K(s', s) \supset \textbf{Know}(\textbf{CanExec}(Achieve(\phi), \sigma, now), do(a, s')).$$

Thus

$$(\forall s')K(s', s) \supset \textbf{SKH}(Achieve(\phi), do(a, s')).$$

Thus

$$(\forall s')K(s', s) \supset \textbf{Can}(\phi, do(a, s')).$$

Thus **Know**$(\textbf{Can}(\phi, do(a, now)), s)$. ■

A.2 An Iterative Definition of Ability

Let us consider a special case of **Can** when the goal can be brought about in a fixed number of steps. We believe this is the case for many applications.

For any non-negative number k, we write $\mathbf{Can}^k_I(k, \phi, s)$ if the agent is able to achieve ϕ in k steps. Inductively, we define

$$\mathbf{Can}^k_I(0, \phi, s) \stackrel{\text{def}}{=} \mathbf{Know}(\phi, s),$$

$$\mathbf{Can}^k_I(k+1, \phi, s) \stackrel{\text{def}}{=} \exists a\, \mathbf{Know}(\mathbf{Can}^k_I(k, \phi, do(a, now)), s).$$

We can show that for any $k \geq 0$, if $\mathbf{Can}^k_I(k, \phi, s)$, then $\mathbf{Can}(\phi, s)$:

Proposition 17

$$\text{For any } k \geq 0,\ \models \forall s(\mathbf{Can}^k_I(k, \phi, s) \supset \mathbf{Can}(\phi, s)).$$

Proof: We prove the proposition by induction over k. $\mathbf{Can}^0(\phi, s) = \mathbf{Know}(\phi, s)$; by proposition 14, this implies that $\mathbf{Can}(\phi, s)$. Assume that $\mathbf{Can}^i(\phi, s) \supset \mathbf{Can}(\phi, s)$ for all $i < k$; by definition, $\mathbf{Can}^k(\phi, s) = \exists a\, \mathbf{Know}(\mathbf{Can}^{k-1}(\phi, do(a, now), s)$; so by the induction hypothesis, we must have that $\exists a\, \mathbf{Know}(\mathbf{Can}(\phi, do(a, now), s)$; by proposition 15, this implies that $\mathbf{Can}(\phi, s)$. ■

We believe for a very wide variety of cases, if $\mathbf{Can}(\phi, s)$, then $\mathbf{Can}^k_I(k, \phi, s)$ for some k. This is true for both the safe and the omelette examples. However, it fails again on the tree example. Roughly speaking, we have

The agent knows that there is a k such that k chops are sufficient.

but

There is no k such that the agent knows that k chops are sufficient.

The easiest way to visualize this is to imagine that in each situation accessible from S_0, there is fixed finite number of chops that will fell the tree, but that there are accessible situations for every natural number.

A.3 Proof of Proposition 13

Lemma 18

$$During(Achieve(\phi), \sigma, s, s') \equiv$$
$$\mathbf{OnPath}(\sigma, s, s') \land \forall s^*(s \leq s^* < s' \supset \neg\mathbf{Know}(\phi, s^*))$$

$$Do(Achieve(\phi), \sigma, s, s') \equiv During(Achieve(\phi), \sigma, s, s') \land \mathbf{Know}(\phi, s')$$

Lemma 19 $\mathbf{Can}(\phi, s) \supset \mathbf{SKH}(Achieve(\phi), s)$

Proof: Suppose that the antecedent holds, that is, that there exists a σ such that **Know**(**CanGet**$(\phi, \sigma, \mathsf{now}), s)$. Take an arbitrary s_s such that $K(s_s, s)$. By the assumption and the definition of **CanGet**, we have that

$$\begin{aligned}&\exists s_e(\mathbf{OnPath}(\sigma, s_s, s_e) \wedge \mathbf{Know}(\phi, s_e) \wedge \\ &\quad \forall s_i[s_s \leq s_i < s_e \supset \exists a\, \mathbf{Know}(\sigma(\mathsf{now}) = a, s_i))]).\end{aligned}$$

Since situations are well founded, we must also have

$$\begin{aligned}&\exists s_e(\mathbf{OnPath}(\sigma, s_s, s_e) \wedge \mathbf{Know}(\phi, s_e) \wedge \\ &\quad \forall s_i[s_s \leq s_i < s_e \supset \neg\mathbf{Know}(\phi, s_i) \wedge \exists a\, \mathbf{Know}(\sigma(\mathsf{now}) = a, s_i))]).\end{aligned}$$

Thus by lemma 18, $\exists s_e\, Do(Achieve(\phi), \sigma, s_s, s_e)$. Take arbitrary s_i, s'_s and s'_i such that $s_s \leq s_i < s_e$, $K(s'_s, s_s)$, $K(s'_i, s_i)$, and $s'_s \leq s'_i$. By the above and positive introspection, it follows that if $s_i = s_e$ then **Know**(ϕ, s'_i). By the above and negative introspection, it follows that if $s_i \neq s_e$ then $\neg$**Know**(ϕ, s'_i). As well, by the above, we must have **OnPath**(σ, s'_s, s'_i). Thus by lemma 18, we must have $During(Achieve(\phi), \sigma, s'_s, s'_i)$, and $Do(Achieve(\phi), \sigma, s'_s, s'_i)$ for $s_i = s_e$. Therefore, **CanExec**$(Achieve(\phi), \sigma, s_s)$. ■

Lemma 20 **SKH**$(Achieve(\phi), s) \supset$ **Can**(ϕ, s)

Proof: Suppose that the antecedent holds, that is, that there exists σ such that **Know**(**CanExec**$(Achieve(\phi), \sigma, \mathsf{now}), s)$. Take an arbitrary s_s such that $K(s_s, s)$. The assumption implies that $\exists s_e\, Do(Achieve(\phi), \sigma, s_s, s_e)$. Thus by lemma 18, we have that

$$\exists s_e(\mathbf{OnPath}(\sigma, s_s, s_e) \wedge \mathbf{Know}(\phi, s_e) \wedge \forall s_i[s_s \leq s_i < s_e \supset \neg\mathbf{Know}(\phi, s_i)]).$$

Take an arbitrary s_i such that $s_s \leq s_i < s_e$. Clearly, it must be the case that $\neg Do(Achieve(\phi), \sigma, s_s, s_i)$. Since $K(s_s, s)$, we must also have that $K(s_s, s_s) \wedge K(s_i, s_i)$. This implies that

$$\neg\forall s'_s, s'_i[K(s'_s, s_s) \wedge K(s'_i, s_i) \wedge s'_s \leq s'_i \supset Do(Achieve(\phi), \sigma, s'_s, s'_i)].$$

Thus, by the assumption and the definition of **CanGet**, we have that

$$\exists a \forall s'_s, s'_i[K(s'_s, s_s) \wedge K(s'_i, s_i) \wedge s'_s \leq s'_i \supset During(Achieve(\phi), \sigma, s'_s, do(a, s'_i))].$$

Clearly $\sigma_l(s_i) = a$; so we have $\exists a$**Know**$(\sigma_l(\mathsf{now}) = a, s_i)$. Therefore, we have that **Know**(**CanGet**$(\phi, \sigma_l, \mathsf{now}), s)$. ■