A Model of Contingent Planning for Agent Programming Languages

Yves Lespérance
Dept. of Computer Sci. & Eng.
York University
Toronto, Canada
lesperan@cse.yorku.ca

Giuseppe De Giacomo
Dip. Informatica e Sistemistica
Univer. di Roma "La Sapienza"
Roma, Italy
degiacomo@dis.uniroma1.it

Atalay Nafi Ozgovde
Dept. of Computer Science
University of Toronto
Toronto, Canada
atalay@cs.toronto.edu

ABSTRACT

In this paper, we develop a formal model of planning for an agent that is operating in a dynamic and incompletely known environment. We assume that both the agent's task and the behavior of the agents in the environment are expressed as high-level nondeterministic concurrent programs in some agent programming language (APL). In this context, planning must produce a deterministic conditional plan for the agent that can be successfully executed against all possible executions of the environment program. We handle actions with nondeterministic effects, as well as sensing actions, by treating them as actions that trigger an environmental reaction that is not under the planning agent's control. Our model of contingent planning is specified for a generic APL with a transition semantics. Within this model, we devise a general procedure for computing the contingent plans. We also show how the model can be instantiated in the situation calculus with programs for the agent and the environment expressed in ConGolog, and we describe an implementation of the planning mechanism in this case.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*ntelligent agents, languages and structures*

General Terms

Languages

Keywords

Agent programming languages, contingent planning

1. INTRODUCTION

Most agents operate in dynamic and incompletely known environments. They must use their sensors to acquire the information they need and they must adjust their behavior to cope with the contingencies that arise. A popular approach to designing such agents involves specifying a library of hierarchically structured plans and reactively executing these plans on-line, using sensed information to select plans, monitor their execution, and recover from exceptions/failure. Various *BDI agent programming languages* [8, 11, 14] have been proposed to support this approach, which leads to very responsive agents. But note that these systems do not perform lookahead or planning in the traditional sense; actions are executed as soon as they are selected. Thus the "BDI agent programming" approach works well if good plans can be specified for all objectives that the agent may acquire and all contingencies that may arise. However, there are often too many possible objectives and contingencies for this to be practical. The agent can also search over the available plans by actually executing them, but this only works if choices can be undone.

To address this problem, some proposals have been made to incorporate lookahead planning mechanisms in agent programming languages (APLs). One line of work aimed at this is the *Golog* language [9] and its successors. Golog is a procedural language defined on top of the situation calculus, a predicate logic framework for reasoning about action. In this approach, instead of specifying just a goal, the user provides a description of how to achieve the goal in the form of a *high-level program*. This is typically a sketchy nondeterministic program that incorporates domain-specific knowledge of how to achieve the goal. The actions and predicates used in the program are specified in a situation calculus domain action theory. The Golog interpreter must search to resolve the nondeterministic choices in the program to find a successful execution. In doing this, it reasons about the preconditions and effects of the actions like a classical planner, but in a search space that is constrained by the the program. *ConGolog* [4] is an extension of Golog that supports concurrent processes. IndiGolog [5] is another extension that allows the programmer to control which parts of the program are planned over and supports online sensing and execution monitoring. Some authors have investigated how planning with Golog task specifications can be performed using state-of-the-art planners. [2] develops an approach for compiling Golog-like task specifications together with the associated domain definition into a PDDL 2.1 planning problem that can be solved by any PDDL 2.1 compliant planner. [1] describes techniques for compiling Golog programs that include sensing action into domain descriptions that can be handled by operator-based planners.

Recently, [14, 15] have proposed the CanPlan and CanPlan2 languages, that incorporate a hierarchical task network (HTN) planning mechanism into a classical BDI agent programming language. Earlier less formal work on planning in APLs is reviewed in [14].

Both IndiGolog and CanPlan only generate sequential

Cite as: A Model of Contingent Planning for Agent Programming Languages, Lespérance, Y., De Giacomo, G. and Ozgovde, A.N., *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May,12-16.,2008,Estoril,Portugal,pp.477-484.

plans. They cannot produce conditional plans that branch on the outcome of a sensing action or nondeterministic world-changing action. This is necessary in many applications. In [12, 6, 13], models of planning with incomplete knowledge and sensing actions are developed. [12] discusses how a planning/search construct for IndiGolog can be defined based on its model. However, these accounts of planning assume that the world only changes as a result of the agent's actions and that non-sensing actions are deterministic.

In this paper, we will generalize this work and develop a model of *planning* that deals with a *dynamic environment*. Our approach in doing this will be to model the possible behaviors of the agents in the environment as a nondeterministic concurrent program that runs concurrently with the agent program, but at a higher priority. The planner will search for a deterministic conditional plan such that for all the executions of this conditional plan concurrently with the environment program, the agent's high-level program is successfully executed. We handle actions with nondeterministic effects, as well as sensing actions, by treating them as actions that trigger an environmental reaction that is not under the planning agent's control. Our model of contingent planning is specified for a generic APL with a transition semantics. Within this model, we devise a general procedure for computing the solution conditional plans. We also show how the model can instantiated in the situation calculus with programs for the agent and the environment expressed in ConGolog. We describe an implementation of the planning mechanism in this case.

We should point out that there has been much previous work on conditional/contingent planners. However, almost none of this work deals with procedurally specified tasks/behaviors or how contingent planning might fit in an APL (one exception is [3]). Note that our model is limited to contingent planning. We model other agents as mere nondeterministic processes rather than as rational decision makers (as in game theory). We also do not model differences in the knowledge that different agents may have. This limits the applicability of our approach, but makes it easier to implement.

In the next section, we sketch some examples to illustrate the range of agent-environment interactions that our approach can support. After that, we describe how contingent planning tasks can be specified in a generic APL. Then, we develop our formalization of what is a solution to such a contingent planning problem, and prove some properties about it. After that, we show how the abstract APL semantic model that we used can be concretely specified for ConGolog and the situation calculus. Then, we discuss an implementation of our formalization in Prolog and IndiGolog. Finally, we conclude by reviewing the paper's contributions and discussing open problems.

2. SOME EXAMPLES IN CONGOLOG

Let us show how our contingent planning model works by giving some examples of environments and agent tasks specified as programs. We will express these in ConGolog [4]. This APL provides the following programming constructs:

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

Among these constructs, we notice the presence of of nondeterministic constructs. These include $(\delta_1 \mid \delta_2)$, which nondeterministically chooses between programs δ_1 and δ_2, $\pi\, \vec{x}.\, \delta$, which nondeterministically picks a binding for the variables $\vec{x}$ and performs the program δ for this binding of $\vec{x}$, and δ^*, which performs δ zero or more times. ConGolog also provides constructs for concurrent programming. $(\delta_1 \parallel \delta_2)$ expresses the concurrent execution (interpreted as interleaving) of the programs δ_1 and δ_2. In $(\delta_1 \rangle\!\rangle\, \delta_2)$, δ_1 has higher priority than δ_2, and δ_2 may only execute when δ_1 is done or blocked. An interrupt $\langle\ \vec{x} : \phi \rightarrow \delta\ \rangle$ triggers when for some binding of $\vec{x}$, the condition ϕ holds; then the body δ is executed for this binding; afterwards the interrupt may trigger again. Synchronization can be done by waiting for conditions involving fluents and performing actions that set them. See [4] for a detailed account of ConGolog.

Now, let us sketch some examples of modeling contingencies and environment dynamics, as well as planning agent tasks as ConGolog programs. Assume a typical blocks world domain formalized in the usual way, and a planning agent PA that wants to build a tower. Our first example involves an interfering agent IA that sometimes moves blocks that the planning agent has stacked back to the table. We could specify IA's behavior as follows:

```
proc $interferingAgtBehavior(IA, n)$
  $(n \leq 0?) \mid$
  $((n > 0 \land LastActionNotBy(IA) \land \exists x, y On(x, y))?\ ;$
    $[\pi\, x. \exists y On(x, y)?; moveToTable(IA, x);$
      $interferingAgtBehavior(IA, n-1)] \mid$
    $[noOp(IA);\ interferingAgtBehavior(IA, n)])$
endProc
```

Note that the n parameter is a bound on the number of interfering $moveToTable$ actions that can be performed by the IA agent; without such a bound, it would be impossible for our planning agent to devise a plan that would achieve having a tower for all executions of the environment.

The planning agent's task of building a three blocks tower could be modeled by the following program:

```
proc $mkTower(PA)$
    while $\neg HaveTower$ do
      if $\exists x, y\, On(x, y)$ then
        $\pi\, x, z.[\exists y\, On(x, y)?; move(PA, z, x)]$
      else
        $\pi\, x, y.move(PA, x, y)$
      endIf
    endWhile
endProc
```

Here, the agent first tries to stack two blocks and then to stack a third block on top of these. There is only a limited amount of nondeterminism in this task specification, i.e. the choice of which block to move. We can devise a conditional plan to perform this task against the interfering agent IA with a set bound on the number of interfering moves.

We could also use a generic planning procedure to specify the planning agent's behavior, leaving all choices to the planning mechanism:

```
proc $genericPlanner(agt, n, m)$
  $(n \leq m)?$;
  $([actionSequence(agt, 0, n); Goal?]$ |
   $genericPlanner(agt, n+1, m))$
endProc
```

```
proc $actionSequence(agt, n, m)$
  $(n = m)?$ |
  $[(n \neq m)?; \pi a.[primitiveAction(a)?; a]$;
   if $LastActionNotBy(agt)$
    then $genericPlanner(agt, 0, m-n-1)$
    else $actionSequence(agt, n+1, m)$ endIf]
endProc
```

The $Goal$ predicate must be defined by the user, in the case of our example $Goal \overset{\text{def}}{=} HaveTower$. This $genericPlanner(agt, n, m)$ procedure performs iterative deepening search for a conditional plan that achieves $Goal$. The procedure uses a bound n on the depth of the conditional plan. The procedure is first called with $n = 0$, and if it fails with this bound, the bound is increased by 1 and the search is run again, until a plan is found or the bound n exceeds the absolute depth limit m. When an exogenous action happens, it is assumed that a branch occurs in the plan at that point and the $genericPlanner$ procedure is called recursively to find a subplan whose depth does not exceed the remaining credit. The procedure minimizes the depth of the plan generated, not the total number of actions it contains.

As a second example, consider a blocks world where the action of moving a block onto another may fail and the moved block may end up on the table. We can model this in a somewhat similar way to the previous example. We can replace the $move(agt, x, y)$ action by an initial $moveAttempt(agt, x, y)$ action and two outcome determining actions performed by a nature agent NA, $moveSucceeds(NA, agt, x, y)$ and $moveFails(NA, PA, x, y)$. The nature agent's behavior can be specified as follows:

```
proc $natureBehavior(NA, n)$
  $\pi\, x, y.[(n > 0 \wedge MoveAttempted(PA, x, y))?$;
    $moveFails(NA, PA, x, y)$;
    $natureBehavior(NA, n-1)]$ |
  $\pi\, x, y.[MoveAttempted(PA, x, y)?$;
    $moveSucceeds(NA, PA, x, y)$;
    $natureBehavior(NA, n)]$
endProc
```

Here as well, we use a parameter n to bound the number of failures so that the goal is achievable.

Our third example involves *sensing*. Imagine that blocks may be wet and that wet blocks cannot be moved. Suppose also that the environment includes a humidity sensor agent HSA that can be queried to find out whether a given block is wet. We could specify the behavior of HSA as follows:

```
proc $humiditySensorBehavior(HSA)$
  $\langle\ x : WetnessQueried(PA, HSA, x) \rightarrow$
    $(reportWet(HSA, PA, x)$ |
     $reportNotWet(HSA, PA, x))\ \rangle$
endProc
```

Here, we assume that there is an action $queryWetness(agt, HSA, x)$ that an agent can use to query the HSA sensor agent and two sensor report actions that the sensor can perform as a response. We assume that these reports must be truthful, i.e. that $reportWet(HSA, agt, x)$ has as precondition $Wet(x)$ and similarly for the "not wet" case. When the planning agent observes the sensor report, it gets to know whether the block was wet by inferring that the report's precondition must have been true.

Note that the planning agent could also acquire knowledge by observing an outcome determining action. For example, if we suppose that move attempts succeed if and only if the moved block is not wet, and make this a precondition of the outcome actions, then the planning agent could perform a move attempt to find out whether a block is wet. In general, whenever the planning agent observes any environment action, it learns that it must have been executable, i.e. learns that its preconditions held (if it did not know this already). Any environment action can be "knowledge producing".

We could also have an additional helpful environment agent DA that dries wet blocks when requested:

```
proc $dryingAgtBehavior(DA)$
  $\langle\ x : DryingRequested(PA, DA, x) \rightarrow dry(DA, x)\ \rangle$
endProc
```

An environment model/program could involve one or several such environment agents running concurrently.

3. CONTINGENT PLANNING IN AN APL

As mentioned earlier, we specify our model of contingent planning for a generic APL. Both the agent's task and the behavior of the agents in the environment will be expressed as nondeterministic concurrent programs in this APL. We will assume that a structural operational semantics (transition system) [10] has been specified for this APL. The details of this semantic specification differ from one APL to another. But we will abstract over these differences and assume that the APL semantics defines a *transition relation* over *agent configurations* $\langle \delta, s \rangle$ that consist of a program δ and a state s. For instance in 3APL [8], the program is a set of "goals" running concurrently and the state consists of a belief state and a ground substitution for variables in the program. In AgentSpeak(L) [11], the program is a set of "intentions" running concurrently and the state is a set of events, a belief base, a set of actions, and a label. In ConGolog, the program is a high-level concurrent program and the state is a ground situation term (together with a fixed situation calculus basic action theory). We assume that the state only changes as a result of the performance of actions (we consider exogenous events to be actions) and that there is a function $actsPerf(s) = \vec{a}$ that for every state s, returns the sequence of actions already performed in that state $\vec{a}$. We also use the notation $actsPerf(s, s')$ to denote the sequence of actions performed in getting from state s to a later state s', i.e. $actsPerf(s, s') = \vec{a}$ iff $actsPerf(s') = actsPerf(s) \circ \vec{a}$ (note that $actsPerf(s, s') = \langle\rangle$ iff $s' = s$). We also assume that actions themselves are deterministic, although the environment can nondeterministically choose between different actions.

To use planning to find a plan/strategy to perform an agent task, we also need a specification of when an agent is in a configuration where its task can be considered successfully completed. We call such a configuration *final* and assume that the set of such *final* configurations has been specified. For instance, CanPlan [14] uses the intention *nil* for this, while ConGolog defines a *Final* predicate.

Another issue arises in connection with incomplete knowledge. When performing its own program/task, the agent should only advance from the current configuration to a new one when it *knows* that there is a legal transition between them. For instance, when executing the program involves performing a primitive action, the agent should know that the action is executable in the current state, and when the program involves testing a condition, it should know whether this condition holds in the current state. Typically, the semantics would require that the action's executability (resp. the tested condition or its negation) be entailed by the agent's belief base for the transition to exist. These epistemic preconditions are appropriate for executing the agent's program, but they are not right when one is considering possible executions of the environment program in a contingent planning context. The plan we produce should specify a suitable agent action in response to any environment action that the agent *considers possible*, i.e. that is *consistent* with what the agent knows. For example, if the environment program is **if** ϕ **then** a_1 **else** a_2 **endIf** and the agent does not know whether the condition ϕ holds (i.e. both ϕ and $\neg\phi$ are consistent with its beliefs), then its plan should prescribe a suitable response to both environment actions a_1 (when ϕ holds) and a_2 (when $\neg\phi$ holds). Existing APL semantics may need to be extended to specify this kind of "consistent" transition relation for environment programs.

Thus in our abstract account, we will rely on the following primitives:

- $\mathit{EnvTrans}(\langle\rho, s\rangle, \langle\rho', s'\rangle)$: the agent considers it possible that the environment program ρ in state s can make a transition to state s' with the program ρ' remaining;
- $\mathit{AgtTrans}(\langle\delta, s\rangle, \langle\delta', s'\rangle)$: the agent knows that the agent program δ in state s can make a transition to state s' with the program δ' remaining;
- $\mathit{AgtFinal}(\langle\delta, s\rangle)$: the agent knows that the agent program δ can legally terminate in state s.

Let us also define a few auxiliary notions in terms of these primitives:

$$\mathit{EnvTrans}(\langle\rho, s\rangle) \overset{\text{def}}{=} \{\langle\rho', s'\rangle) | \mathit{EnvTrans}(\langle\rho, s\rangle, \langle\rho', s'\rangle)\},$$

$$\mathit{EnvBlocked}(\langle\rho, s\rangle) \overset{\text{def}}{=} \mathit{EnvTrans}(\langle\rho, s\rangle) = \emptyset,$$

$$\mathit{FiniteEnvTrans}(\langle\rho, s\rangle) \overset{\text{def}}{=} \mathit{EnvTrans}(\langle\rho, s\rangle) \text{ is a finite set.}$$

Moreover, let $\mathit{EnvTrans}^*(\langle\rho, s\rangle, \langle\rho', s'\rangle)$ be the reflexive transitive closure of *EnvTrans*, i.e. $\mathit{EnvTrans}^*(\langle\rho, s\rangle, \langle\rho', s'\rangle)$ holds iff $\langle\rho', s'\rangle$ is reachable from $\langle\rho, s\rangle$ in 0 or more transitions. Also let $\mathit{EnvTrans}^*(\langle\rho, s\rangle)$ and $\mathit{FiniteEnvTrans}^*(\langle\rho, s\rangle)$ be defined for $\mathit{EnvTrans}^*$ as were $\mathit{EnvTrans}(\langle\rho, s\rangle)$ and $\mathit{FiniteEnvTrans}(\langle\rho, s\rangle)$ for *EnvTrans*. Finally, let

$$\mathit{EnvTrans}^m(\langle\rho, s\rangle, \langle\rho', s'\rangle) \overset{\text{def}}{=} \mathit{EnvTrans}^*(\langle\rho, s\rangle, \langle\rho', s'\rangle) \text{ and } \mathit{EnvBlocked}(\langle\rho', s'\rangle),$$

i.e. $\langle\rho', s'\rangle$ can be reached by executing $\langle\rho, s\rangle$ until it blocks. Also, let $\mathit{EnvTrans}^m(\langle\rho, s\rangle)$ be defined as was $\mathit{EnvTrans}(\langle\rho, s\rangle)$ for *EnvTrans*.

Let us now define what an execution of an agent program in a dynamic environment is. We model the environment's possible behaviors by a nondeterministic program ρ. So for us, a configuration will be a triple $\langle\delta, \rho, s\rangle$ formed by a program for the agent δ, a program for the environment ρ, and a state s. We assume that ρ executes with higher priority than the planning agent's program δ. This seems reasonable in a contingency planning context, since the agent cannot control when the environment acts and we want planning to produce a plan that works for any possible behavior by the environment (different assumptions may be reasonable in other contexts). Both the agent program and environment program may be nondeterministic. We also assume that environment/exogenous actions are fully observable. Since the environment program ρ runs at higher priority, it should eventually block so that the agent can act. We don't require ρ to ever terminate. We say that a configuration $\langle\delta, \rho, s\rangle$ *may evolve* to a configuration $\langle\delta', \rho', s'\rangle$ if and only if either $\mathit{EnvTrans}(\langle\rho, s\rangle, \langle\rho', s'\rangle)$ holds and $\delta' = \delta$, or both $\mathit{EnvBlocked}(\langle\rho, s\rangle)$ and $\mathit{AgtTrans}(\langle\delta, s\rangle, \langle\delta', s'\rangle)$ hold and $\rho' = \rho$. We also say that a configuration $\langle\delta, \rho, s\rangle$ is *final* whenever $\mathit{EnvBlocked}(\langle\rho, s\rangle)$ and $\mathit{AgtFinal}(\langle\delta, s\rangle)$.

Given this, we can now define various kinds of executions. An *execution* of an agent program δ in an environment ρ starting from a state s is a possibly infinite sequence of *configurations* $\langle\delta_0 = \delta, \rho_0 = \rho, s_0 = s\rangle$, $\langle\delta_1, \rho_1, s_1\rangle$, ... such that for all pairs of successive configurations, $\langle\delta_i, \rho_i, s\rangle$ may evolve to its successor $\langle\delta_{i+1}, \rho_{i+1}, s_{i+1}\rangle$. A finite execution is *complete* if and only if its last configuration is either final or there is no configuration that it may evolve to. In the former case, we say that the execution *successfully terminates*; in the latter case, we say that the execution is *stuck* or has reached a *dead-end*. We also say that a complete execution *ends in* state s_n if s_n is the state of its last configuration.

Let us now discuss a few issues related to environment modeling. Sometimes, we want to model cases where the environment may do an action or do nothing. Since we assume that the environment program is running at higher priority, the "do nothing" case must be represented by an explicit "no op" transition. For example, a case where the environment may either do action a or do nothing and then waits until P becomes true before continuing must be modeled as:

$$(a|noOp); P?)); \ldots$$

where *noOp* is a primitive action that has no effects on any fluent. If we left out the *noOp*,, i.e.

$$(a; P?); \ldots$$

then the only possible execution would be for the environment to do a, since it runs at higher priority and the agent cannot do a transition when the environment can.

A good environment program/model should also be such that in any configuration, one knows what the environment may do next. That is, after having observed a particular sequence of actions, there should be only one possible remaining environment program. For example, the environment program

$$((a; P?; b)|(a; P?; c))$$

is a bad model because after observing a, the remaining environment program may be either $(P?; b)$ or $(P?; c)$, i.e. either the environment may only do b or it may only do c

and the planner has no way of knowing which. The program

$$a; P?; (b|c)$$

does not have this problem. To rule this out, we impose the following constraint:

> for any configuration $\langle \delta, \rho, s\rangle$ in an execution that started in the initial configuration $\langle \delta_i, \rho_i, s_i\rangle$, it must be the case that if $EnvTrans(\langle \rho, s\rangle, \langle \rho', s'\rangle)$, $EnvTrans(\langle \rho, s\rangle, \langle \rho'', s''\rangle)$, and $actsPerf(s'') = actsPerf(s')$, then $\rho' = \rho''$.

Another case of a problematic environment model is the program $P?; a$ when both P and $\neg P$ are consistent with the agent's beliefs. Here it is consistent that the environment does a when P holds and it is also consistent that it does nothing when $\neg P$ holds. In the latter case, the model should make it explicit that the environment chooses the second option by making a transition. So we should use the environment program

$$\textbf{if } P \textbf{ then } a \textbf{ else } noOp \textbf{ endIf}$$

instead of the one above. To rule out this kind of problematic environment model, we impose the following constraint:

> for any configuration $\langle \delta, \rho, s\rangle$ in an execution that started in the initial configuration $\langle \delta_i, \rho_i, s_i\rangle$, it must be the case that if there exist ρ', s' such that $EnvTrans(\langle \rho, s\rangle, \langle \rho', s'\rangle)$, then $EnvNotBlocked(\langle \rho, s\rangle)$, where $EnvNotBlocked(\langle \rho, s\rangle)$ is a new semantic primitive meaning that the agent knows in state s that the environment program ρ is not blocked.

These constraints limit the use of tests, whose execution is unobservable, in environment programs.

4. FORMALIZATION

Informally, an agent is *able/knows how* to execute a program δ in a state s where the environment behaves as specified by the program ρ if the agent is able to repeatedly choose some action that is known to be executable and allowed by her program, such that no matter what exogenous actions occur (as allowed by the environment program ρ), she can continue this process with what remains of her program and eventually reach a configuration where she knows that she can legally terminate.

We can formalize $Able(\delta, \rho, s)$, i.e., that the agent is able to execute the program δ in an environment that behaves as specified by the program ρ in state s as follows. Let $Able(\delta, \rho, s)$ be the smallest relation $\mathcal{R}(\delta, \rho, s)$ such that:

```
(A) for all triples $(\delta, \rho, s)$, if
        $EnvBlocked(\langle \rho, s\rangle)$ and $AgtFinal(\langle \delta, s\rangle)$,
    then $\mathcal{R}(\delta, \rho, s)$;

(B) for all triples $(\delta, \rho, s)$, if
        $EnvBlocked(\langle \rho, s\rangle)$ and
      there exist $\delta', s'$ such that
          $AgtTrans(\langle \delta, s\rangle, \langle \delta', s'\rangle)$ and $\mathcal{R}(\delta', \rho, s')$,
    then $\mathcal{R}(\delta, \rho, s)$;

(C) for all triples $(\delta, \rho, s)$, if
        it is not the case that $EnvBlocked(\langle \rho, s\rangle)$ and
      for all $\rho', s'$,
          $EnvTrans(\langle \rho, s\rangle, \langle \rho', s'\rangle)$ implies $\mathcal{R}(\delta, \rho', s')$,
    then $\mathcal{R}(\delta, \rho, s)$.
```

Nothing in the above ensures that there exists a *finitely branching* conditional plan for the agent in case (C). We can strengthen the definition to get this by replacing (C) with the following:

```
(C') for all triples $(\delta, \rho, s)$, if
         it is not the case that $EnvBlocked(\langle \rho, s\rangle)$ and
        $FiniteEnvTrans^*(\langle \rho, s\rangle)$ and
        for all $\rho', s'$,
            $EnvTrans^m(\langle \rho, s\rangle, \langle \rho', s'\rangle)$ implies $\mathcal{R}(\delta, \rho', s')$,
     then $\mathcal{R}(\delta, \rho, s)$,
```

Let's call this stronger definition $Able^+$

If $Able^+(\delta, \rho, s)$ holds, it is straightforward to generate a conditional plan that the agent can follow to successfully execute its program no matter how the environment behaves, given that the agent can always choose an action that leads to successful termination and that environment actions are fully observable. We write $AbleBy(\sigma, \delta, \rho, s)$ to mean that the agent is able to execute program δ in an environment that behaves as specified by program ρ in state s by executing the conditional program σ. Formally, let $AbleBy(\sigma, \delta, \rho, s)$ be the smallest relation $\mathcal{R}(\sigma, \delta, \rho, s)$ such that:

```
(A) for all triples $(\delta, \rho, s)$, if
        $EnvBlocked(\langle \rho, s\rangle)$ and $AgtFinal(\langle \delta, s\rangle)$,
    then $\mathcal{R}(nil, \delta, \rho, s)$;

(B) for all quadruples $(\sigma, \delta, \rho, s)$, if
        $EnvBlocked(\langle \rho, s\rangle)$ and
        there exist $\delta', s'$ such that
            $AgtTrans(\langle \delta, s\rangle, \langle \delta', s'\rangle)$ and $\mathcal{R}(\sigma, \delta', \rho, s')$ and
            $actsPerf(s, s') = \langle\rangle$,
    then $\mathcal{R}(\sigma, \delta, \rho, s)$;

(C) for all $a, \sigma, \delta, \rho, s$, if
        $EnvBlocked(\langle \rho, s\rangle)$ and
        there exist $\delta', s'$ such that
            $AgtTrans(\langle \delta, s\rangle, \langle \delta', s'\rangle)$ and $\mathcal{R}(\sigma, \delta', \rho, s')$ and
            $actsPerf(s, s') = a$,
    then $\mathcal{R}((a; \sigma), \delta, \rho, s)$;

(D) for all triples $(\delta, \rho, s)$, if
         it is not the case that $EnvBlocked(\langle \rho, s\rangle)$ and
        $FiniteEnvTrans^*(\langle \rho, s\rangle)$ and
        for all $\rho', s'$
            $EnvTrans^m(\langle \rho, s\rangle, \langle \rho', s'\rangle)$ implies
                for some $\sigma'$, $\mathcal{R}(\sigma', \delta, \rho', s')$,
    then $\mathcal{R}(\sigma, \delta, \rho, s)$ where
        $\sigma =$ if $Done(actsPerf(s, s_1))$ then $\sigma_1$ else
            ...
                if $Done(actsPerf(s, s_n))$ then $\sigma_n$ else $nil$
        and $EnvTrans^m(\langle \rho, s\rangle) = \{\langle \rho_1, s_1\rangle, \ldots, \langle \rho_n, s_n\rangle\}$
        and $\mathcal{R}(\sigma_i, \delta, \rho_i, s_i)$ for $i = 1, \ldots, n$.
```

Note that in the above, we assume that the programming language allows testing whether a given sequence of actions $\vec{a}$ has just been done using the condition $Done(\vec{a})$.

5. PROPERTIES

We can prove that whenever the agent is able to execute a program δ in an environment behaving as ρ in state s, i.e. $Able^+(\delta, \rho, s)$ holds, then there exist a conditional program σ such that the agent is able to execute the program δ in an environment behaving as ρ in state s by executing σ, i.e. $AbleBy(\sigma, \delta, \rho, s)$ also holds:

Theorem 1. *If* $Able^{+}(\delta, \rho, s)$*, then there exists* σ *such that* $AbleBy(\sigma, \delta, \rho, s)$*.*

We can also prove the following:

Theorem 2. *If* $AbleBy(\sigma, \delta, \rho, s)$*, then* $Able^{+}(\delta, \rho, s)$*.*

It is well known that there are goals that are achievable, but cannot be achieved by any conditional program, for instance, goals that require unbounded iteration. Similarly, there are programs that an agent is able to execute, but where there is no conditional plan that can be followed to execute the program. So the definitions of ability proposed here only tell whether the program can be executed by following a conditional plan. This issue is discussed in [12, 6, 13], and the authors give definitions of ability that allow arbitrary deterministic plans (including iterative ones) as potential solutions, in the context of a static environment. We believe that these definitions can be adapted to handle contingent planning/dynamic environments. But we leave this for future work, as we want to focus on the conditional planning case here.

We can also show that if the agent is able to execute program δ in an environment behaving as ρ in state s by executing the conditional program σ, i.e. $AbleBy(\sigma, \delta, \rho, s)$, then σ is guaranteed to successfully terminate in a state where δ can also terminate no matter how the environment acts:

Theorem 3. *If* $AbleBy(\sigma, \delta, \rho, s)$*, then*

1. *every execution of* σ *in environment* ρ *starting in state* s *successfully terminates, and*

2. *for every complete execution of* σ *in environment* ρ *starting in state* s*, if this execution ends in state* s'*, then there exists an execution of* δ *in environment* ρ *starting in state* s *that successfully terminates in* s'*.*

6. CONCRETE SPECIFICATION IN CONGOLOG

We can ground the abstract account of contingent planning developed earlier by defining the primitives that it uses in terms of an existing APL semantics. In this section, we do this for ConGolog.

ConGolog's semantics is a structural operational semantics (with a single-step transition relation) in the style of [10]. This semantics introduces two special predicates $Trans$ and $Final$: $Trans(\delta, s, \delta, s')$ means that by executing program δ 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. For example, the transition requirements for sequence are

$$\begin{aligned} &Trans([\delta_1;\delta_2], s, \delta', s') \equiv \\ &\quad Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s') \vee \\ &\quad\quad \exists \delta''. Trans(\delta_1, s, \delta'', s') \wedge \delta' = (\delta''; \delta_2) \end{aligned}$$

i.e., to single-step the program $(\delta_1;\delta_2)$, either δ_1 terminates and we single-step δ_2, or we single-step δ_1 leaving some δ'', and $(\delta'';\delta_2)$ is what is left of the sequence. We denote the set of ConGolog semantic axioms by $\mathcal{C}$.

As mentioned in Section 2, when the planning agent observes an environment action, it may acquire additional knowledge. At a minimum, it learns that the environment action must have been executable, i.e. that its preconditions must have been true (assuming the agent did not know this already). Let us represent the new information acquired by the planning agent in a situation s by $NewInfo(s)$. For now, let's assume that the only new information acquired by the planning agent as it observes environment actions is that they were executable. Then we can use the definition:[1]

$$NewInfo(s) \overset{\text{def}}{=} executable(s), \text{ where}$$

$$executable(s) \overset{\text{def}}{=} \forall a, s. S_0 < do(a, s) \leq s \supset Poss(a, s).$$

In the remainder, we will handle the knowledge producing effects of observing the actions in the situation s by adding $NewInfo(s)$ to the agent's basic action theory $\mathcal{D}$.

First, we define $EnvTrans(\langle \rho, s\rangle, \langle \rho, s'\rangle)$, i.e. that the agent considers it possible that the program δ in situation s can make a transition to situation s' with the program δ' remaining, as follows:

$$\begin{aligned} &EnvTrans(\langle \rho, s\rangle, \langle \rho, s'\rangle) \overset{\text{def}}{=} \\ &\quad \mathcal{D} \cup \mathcal{C} \cup \{NewInfo(do(actsPerf(s), S_0))\} \cup \\ &\quad \{Trans(\rho, do(actsPerf(s), S_0), \rho', do(actsPerf(s'), S_0))\} \\ &\quad \text{is consistent.} \end{aligned}$$

Note that we map the state s into the situation term $do(actsPerf(s), S_0)$, which stands for the situation where the actions performed in state s have occurred. We also add $NewInfo(do(actsPerf(s), S_0))$, i.e. $executable(do(actsPerf(s), S_0))$, to the theory (the agent's knowledge) at every step. This captures the knowledge expansion that may take place when the agent observes the occurrence of exogenous actions.

Secondly, we define $AgtTrans(\langle \delta, s\rangle, \langle \delta', s'\rangle)$, i.e. that the agent knows that the program δ in state s can make a transition to state s' with the program δ' remaining as follows:

$$\begin{aligned} &AgtTrans(\langle \delta, s\rangle, \langle \delta', s'\rangle) \overset{\text{def}}{=} \\ &\quad \mathcal{D} \cup \mathcal{C} \cup \{NewInfo(do(actsPerf(s), S_0))\} \models \\ &\quad\quad Trans(\delta, do(actsPerf(s), S_0), \delta', do(actsPerf(s'), S_0)). \end{aligned}$$

Thirdly, we define $AgtFinal(\langle \delta, s\rangle)$, i.e. that the agent knows that the program δ can legally terminate in state s as follows:

$$\begin{aligned} &AgtFinal(\langle \delta, s\rangle) \overset{\text{def}}{=} \\ &\quad \mathcal{D} \cup \mathcal{C} \cup \{NewInfo(do(actsPerf(s), S_0))\} \models \\ &\quad\quad Final(\delta, do(actsPerf(s), S_0)). \end{aligned}$$

Finally, we define $EnvNotBlocked(\langle \rho, s\rangle)$, i.e. that the agent knows that the environment program ρ is not blocked in state s as follows:

$$\begin{aligned} &EnvNotBlocked(\langle \rho, s\rangle) \overset{\text{def}}{=} \\ &\quad \mathcal{D} \cup \mathcal{C} \cup \{NewInfo(do(actsPerf(s), S_0))\} \models \\ &\quad\quad \exists \rho', s'\, Trans(\rho, do(actsPerf(s), S_0), \rho', s'). \end{aligned}$$

Note that we assume that the action theory $\mathcal{D}$ includes a domain closure axiom for primitive actions. Thus if $\mathcal{D} \cup \mathcal{C} \models \exists \delta', s'\, Trans(\delta, s, \delta', s')$, then there exist ground terms δ', s' such that $\mathcal{D} \cup \mathcal{C} \models Trans(\delta, s, \delta', s')$. For instance, this

[1]We can assume that the agent knows that the situation where the program started was executable and that any action it performed in getting to s was also executable; so the only part of $NewInfo(s)$ that is really new is that the exogenous actions were possible.

is necessary to rule out having both $\mathit{EnvBlocked}(\rho, s)$ and $\mathcal{D} \cup \mathcal{C} \models \exists \rho', s'\, \mathit{Trans}(\rho, do(\mathit{actsPerf}(s), S_0), \rho', s')$.

In the above, we assumed that the only new information acquired by the agent was that the observed environment actions were executable. This may be overly restrictive, as we also assume that the planner knows what program the environment is executing. This environment program may specify that some actions can only occur under some conditions. For example, we may have **if** P **then** a **else** b **endIf** as environment program. Suppose that the planning agent does not know whether P holds initially. Upon observing the action a, the planning agent learns that P must have been true, since it knows that the environment only performs a when P holds. We can handle this by redefining *NewInfo* as follows:

$$\mathit{NewInfo}(\delta_i, \rho_i, s_i, \delta, \rho, s) \overset{\text{def}}{=} \mathit{executable}(s_i) \wedge \mathit{Trans}^*((\rho_i \rangle\!\rangle\, \delta_i), s_i, (\rho \rangle\!\rangle\, \delta), s)$$

The "new information" now depends on several additional parameters: the initial agent program δ_i, initial environment program ρ_i, situation where their execution started s_i, current agent program δ, and current environment program ρ. The new definition says that the planning agent learns that there must have been a partial execution of the environment program together with the agent program from s_i to the current situation s, with all that this implies (for instance, that P must have been true in the above example). Note that the new definition implies the old one. The new parameters in *NewInfo* must be added to all our primitives, for instance, $\mathit{EnvTrans}(\langle \delta_i, \rho_i, s_i\rangle, \langle \delta, \rho, s\rangle, \langle \rho', s'\rangle)$, meaning that the agent considers it possible that the environment program ρ in state s can make a transition to program ρ' and state s' after observing transitions from the initial configuration $\langle \delta_i, \rho_i, s_i\rangle$ to the current configuration $\langle \delta, \rho, s\rangle$, and similarly for the other primitives. The grounding for these can be exactly as before, but using the revised definition of *NewInfo*. For our notions of ability, we now write for instance $\mathit{AbleBy}(\sigma, \delta_i, \rho_i, s_i, \delta, \rho, s)$ to mean that the agent is able to execute the program δ in an environment that behaves as specified by the program ρ in state s by executing the conditional program σ after reaching configuration $\langle \delta, \rho, s\rangle$ from the initial configuration $\langle \rho_i, \delta_i, s_i\rangle$ (and similarly for $\mathit{Able}(\delta_i, \rho_i, s_i, \delta, \rho, s)$). The definitions for these can be exactly as before, but using the revised versions of the primitives with the stronger knowledge expansion. Finally, we can define: $\mathit{AbleBy}(\sigma, \delta, \rho, s) \overset{\text{def}}{=} \mathit{AbleBy}(\sigma, \delta, \rho, s, \delta, \rho, s)$.

7. AN IMPLEMENTATION

We show in this section that it is easy to extract from the above definition a Prolog program that given a program `p` for the agent, a program `e` for the environment, and a starting state `s`, deliberates and returns a conditional plan `c` that allows the agent to execute its program in the environment, i.e., such that $\mathit{AbleBy}(\mathtt{c}, \mathtt{p}, \mathtt{e}, \mathtt{s})$ holds.

```
% Case(A)
ableBy([],P,E,S) :-
   envBlocked(E,S), agtFinal(P,S).
% Case(B)
ableBy(C,P,E,S) :-
   envBlocked(E,S), agtTrans(P,S,P1,S1),
   actsPerf(S,S1,[]), ableBy(C,P1,E,S).
% Case(C)
ableBy([A|C],P,E,S) :-
   envBlocked(E,S), agtTrans(P,S,P1,S1),
   actsPerf(S,S1,[A]), ableBy(C,P1,E,S1).
% Case(D)
ableBy(C,P,E,S) :-
   findall([E1,S1], envTransP(E,S,E1,S1), L),
   ableByBranch(C,P,S,L).

ableByBranch(?(fail),P,S,[]).
ableByBranch(if(done(As),C1,C2),P,S,[[E1,S1]|L]):-
   actsPerf(S,S1,As), ableBy(C1,P,E1,S1),
   ableByBranch(C2,P,S,L).

envBlocked(E,S) :-
   not envTransExists(E,S).
envTransExists(E,S) :-
   envTrans(E,S,E1,S1).

envTransP(E,S,E1,S1) :-
   envTransRTC(E,S,E1,S1), envBlocked(E1,S1).

% Reflexive Transitive closure of envTrans
envTransRTC(E,S,E,S).
envTransRTC(E,S,E2,S2) :-
   envTrans(E,S,E1,S1), envTransRTC(E1,S1,E2,S2).
```

This program relies on the availability of the predicates `agtTrans`, `agtFinal`, `envTrans`, and `actsPerf`. The latter simply extract the actions `As` that where performed in moving from state `S` to `S1`. `agtTrans` and `agtFinal` represent *AgtTrans* and *AgtFinal* respectively, while `envTrans` represents *EnvTrans*.

THEOREM 4. *It* `agtTrans` *and* `agtFinal` *are a* sound *implementation of AgtTrans and AgtFinal, and* `envTrans` *is a* complete *implementation of EnvTrans, then* `ableBy` *is a sound implementation of AbleBy, i.e., for all* `(c,p,e,s)` *such that* `ableBy(c,p,e,s)` *succeeds, AbleBy*$(\mathtt{c}, \mathtt{p}, \mathtt{e}, \mathtt{s})$ *holds.*

We have developed a Prolog implementation of `agtTrans`, `agtFinal` and `envTrans` in the context of the situation calculus and ConGolog. This implementation is based on a recent version of IndiGolog that supports a limited form of incomplete knowledge [16]. Specifically, incomplete knowledge is restricted to having a set of possible values for each functional fluent, and *formulas* can be *possibly true* or *certainly true* (i.e. *known to be true*). Informally, a formula ϕ is *possibly true* if there exists a choice of possible values for the fluents in the formula that makes it true. A formula ϕ is *certainly true*, if $\neg\phi$ is not possibly true, i.e. if every choice of possible values for the fluents in ϕ makes ϕ true.

Now, using the the notion of a formula being certainly true we can devise a sound (though possibly not complete) definition of `agtTrans` and `agtFinal`, while using the notion of a formula being possibly true, we can devise a complete (though possibly not sound) definition of `envTrans`.

In our implementation, we handle the knowledge producing effects of observing environment actions, more specifically learning that these actions' preconditions were true, by adapting the regression mechanism of [16] (which only deals with sensing actions performed by the agent). One provides specifications of preconditions for environment actions of the form `poss_set(a, f, v, w)` and `poss_rej(a, f, v, w)`, meaning that the occurrence of the exogenous action `a` is possible if and only if the fluent `f` has (resp. does not have) the value `v` when condition `w` holds. The regression mechanism uses these to update the set of possible values of fluents. We do not yet handle new information that comes from knowing the environment program.

8. CONCLUSION

Most agents operate in dynamic and incompletely known environments, and they should anticipate relevant contingencies in making plans of action. In this paper, we developed a formal model of contingent planning for use in agent programming languages. We showed how agent tasks and environment agent behaviors can be specified in an APL with a transition semantics and how this can be viewed as a specification of a contingent planning problem. Then we developed a formalization of when a conditional plan is a solution to such a contingent planning problem. We proved that our formalization had some desirable properties. Then, we showed how the abstract APL semantic model that we used in formalizing contingent planning problems and their solutions could be concretely specified for ConGolog and the situation calculus. This confirms the soundness of the model and provides one path to implementation. We then discussed the main elements of an implementation in Prolog, relying on IndiGolog's support for modeling some forms of incomplete knowledge and how it evolves [16].

We should point out that our framework only involved contingent planning, where other agents are modeled as nondeterministic processes rather than as rational decision makers. This limits the applicability of our approach, but makes it more readily implementable. Developing a game-theoretic planning model for agent programming would certainly be a worthy objective (see [7] for one approach to this). Representing and reasoning efficiently about the (incomplete) knowledge that multiple agents have about the world and each other remains a key problem for multiagent planning.

Many other avenues remain for future work. It would be worthwhile to examine whether our form of contingent planning with procedural specifications for the agent's task and the behavior of environment agents can be compiled into the kind of contingent planning problems that operator-based contingent planners can solve, and if so to develop a compilation mechanism, as done in [1] for planning with sensing. Our approach to representing dynamic incompletely known environments should be tested on some realistic applications, to see whether it is sufficiently expressive and easy to use. Perhaps some language enhancements can be helpful for this. We would also like to do more experimentation with our implementation. Finally, it would be interesting to generalize our formalization to support iterative plans.

9. REFERENCES

[1] J. Baier and S. McIlraith. On planning with programs that sense. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06)*, pages 492–502, Lake District, UK, June 2006.

[2] J. A. Baier, C. Fritz, and S. A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, Providence, Rhode Island, USA, September 22 - 26 2007.

[3] A. Bouguerra and L. Karlsson. PC-SHOP: A probabilistic-conditional hierarchical task planner. *Intelligenza Artificiale*, 2(4):44–50, 2005.

[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 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.

[6] G. De Giacomo, S. Sardiña, Y. Lespérance, and H. J. Levesque. On deliberation under incomplete information and the inadequacy of entailment and consistency-based formalizations. In *Working Notes of the 1st Int. Workshop on Programming Multiagent Systems (PROMAS-2003)*, Melbourne, July 2003.

[7] A. Farinelli, A. Finzi, and T. Lukasiewicz. Team programming in Golog under partial observability. In M. M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2097–2102, 2007.

[8] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. J. C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2:357–401, 1999.

[9] 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.

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

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

[12] S. Sardiña, G. De Giacomo, Y. Lespérance, and H. Levesque. On the semantics of deliberation in IndiGolog – from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):259–299.

[13] S. Sardiña, G. De Giacomo, Y. Lespérance, and H. J. Levesque. On the limits of planning over belief states under strict uncertainty. In P. Doherty, J. Mylopoulos, and C. A. Welty, editors, *Principles of Knowledge Representation and Reasoning, Proc. of the 10th Int. Conf. (KR2006)*, pages 463–471, Windemere, UK, June 2006. AAAI Press.

[14] S. Sardiña, L. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. of the 5th Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1001–1008, Hakodate, Japan, May 2006. ACM Press.

[15] S. Sardiña and L. Padgham. Goals in the context of BDI plan failure and planning. In *Proc. of the 6th Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS'07)*, pages 16–24, Honolulu, HI, May 2007. Research Publishing Services.

[16] S. Sardiña and S. Vassos. The Wumpus World in IndiGolog: A preliminary report. In *Working Notes of the 6th Workshop on Nonmonotonic Reasoning, Action, and Change (at IJCAI-05)*, Edinburgh, 2005.