A Semantics for HTN Methods

Robert P. Goldman
SIFT, LLC
211 N. First St.
Minneapolis, MN 55401
rpgoldman@sift.info

Abstract

Despite the extensive development of first-principles planning in recent years, planning applications are still primarily developed using knowledge-based planners which can exploit domain-specific heuristics and weaker domain models. Hierarchical Task Network (HTN) planners capture domain-specific heuristics for more efficient search, accommodate incomplete causal models, and can be used to enforce standard operating procedures. Unfortunately, we do not have semantics for the methods or tasks that make up HTN models, that help evaluate the correctness of methods, or to build a reliable executive for HTN plans. This paper fills the gap by providing a well-defined semantics for the methods and plans of SHOP2, a state-of-the-art HTN planner. The semantics are defined in terms of concurrent golog (ConGolog) and the situation calculus. We provide a proof of equivalence between the plans generated by SHOP2 and the action sequences of the ConGolog semantics. We show how the semantics reflects the distinction between plan-time and execution-time, and provide some simple examples showing how the semantics can support method verification. The semantics provide an implementation-neutral specification for an executive, showing how an executive must treat the plans SHOP2 generates in order to enforce the expected behaviors. Future directions include automated verification of method specifications, automatically generating plan monitors, and plan revision and repair.

Introduction

Despite extensive development of domain independent, first-principles planning in recent years, *knowledge-based planning* that exploits domain-specific heuristics is still the solution of choice for most planning applications. As a method for knowledge-based planning, Hierarchical Task Network (HTN) planning has much to recommend it. HTN planners capture domain-specific heuristic information in their *methods*, which are skeletal procedures that indicate how a task may be performed. HTN methods simplify knowledge engineering by avoiding the need for full causal (precondition and postcondition) models, and allowing designers to stipulate that certain actions will be performed "just because I say so." The methods of an HTN planner can easily encode constraints on the *means* by which a goal may be achieved. Such constraints can only be engineered clumsily in first-principles planning domains.[1]

In many applications, standard operating procedures must be obeyed or complete causal models are difficult to obtain, so applying first-principles planners is difficult. For example, we may know that a particular treatment in a medical domain is associated with good outcomes, and need to enforce it, without being able to provide a causal account sufficient to convince a first-principles planner to derive the desired plan fragment. Other domain-specific knowledge may include constraints relating to possible futures that are not certain but should influence plans, such as retaining a fuel reserve for unmodeled contingencies. It is easy to encode such standard procedures and operational constraints in HTN methods. While in theory HTN planning is harder than conventional planning because the HTN framework is more expressive, in practice HTNs achieve efficiency through the heuristic information in their methods.

HTNs are particularly useful in planning for autonomous systems. When planning for autonomous systems, one typically marries a projective planner with a smart, reactive executive. The projective planner gives the autonomous system a broad perspective, notably over long term state trajectories, resource usage, and optimization. These domains are often very complex and incompletely modeled. With the aid of domain knowledge, one may be able to tackle specific cases of planning problems whose general cases are still beyond the state of the art.

Unfortunately, we don't have clear semantics for HTN methods and plans, making it extremely difficult to evaluate the correctness of methods, or to build a reliable executive for HTN plans. This paper fills the gap by providing a well-defined semantics for the methods and plans of SHOP2, a state of the art HTN planner. The semantics are defined using concurrent golog (ConGolog) and the situation calculus.

The semantics will help us ask, for any given method and task, "is this a good method for this task?" The soundness of SHOP2 provides a guarantee that no method will expand to an illegal plan, but makes no guarantees beyond that. To go beyond that, we must reason about what kind of plans will be produced when using a given HTN method. In particular,

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] E.g., Boddy et al. (2005) describe efforts to encode standard plan snippets into PDDL for cyber security.

we want to be able to postulate invariants — properties that should hold of any plan containing the method — and use our semantics to evaluate these invariants.

Our new HTN semantics also enable us to better develop *protections* for HTN plans. Protections are a mechanism used in HTNs to avoid the introduction of interference into the body of a method. A protection will be added by some HTN method or action, and the HTN algorithm will not introduce any actions that would violate the protection until it has been removed. For example, in a satellite control application, one might have a "downlink" method that would point the satellite's antenna at the ground station, protect the proposition `(point-at antenna ground-station)`, carry out some additional actions to transmit data, terminate the transmission, and then remove the protection. The protection would keep the planner from inadvertently introducing an operator that would change the antenna's orientation. As discussed below, our semantics for HTN methods give a means to identify useful protections.

Plan semantics help us in managing the *execution* of HTN plans. Our HTN semantics provide an implementation-neutral specification for an executive, showing how an executive must treat the plans that SHOP2 generates in order to enforce the expected behaviors. We will also show that the semantics allow the planner to tell the executive, based on a model of exogenous actions (or disturbances), what conditions it should monitor during execution.

Our work here is in the tradition of work on verifying and validating conventional programs. However, it is important to stress that the goal here is in many ways more difficult than analyzing programs. Although it is tempting to treat HTN methods as if they were procedures, they are not, because they are not executed as they are written. Instead, they pass through a complex planning process which selects and composes the HTN methods, potentially intertwining multiple methods' task networks. It is tempting to overlook this distinction, treating planner methods as procedures, and confusing *projecting* the effects of actions with *executing* those actions. We try to clarify those issues here.

In the next section we briefly introduce the HTN planner SHOP2, and sketch its planning algorithm. We discuss some previous work that addresses issues related to ours, then present concurrent Golog, a procedural extension of the situation calculus, and show how it can provide a semantics for SHOP2 plans. We next address issues that arise from the distinction between plan-time and execution-time reasoning while planning, and conclude by suggesting some future directions opened by this work.

SHOP2: Simple Hierarchical Ordered Planner

SHOP2 is a modern HTN planner with a relatively simple, clean implementation, that is easy to adapt for applications, and that has performed well in past planning competitions (Nau, Au, and others 2003).Another advantage of the SHOP2 planning system is that it is available under a generous open-source license, and is maintained at SourceForge[2]. SHOP2 has made HTN planning more easily open to experimentation than the dominant past HTN planners, Sipe (Wilkins 1988) and O-Plan (Currie and Tate 1991), which for all of their considerable advantages, are enormously complex and proprietary. We have discussed elsewhere application-based reasons for our preference for HTN over first-principles planning(Goldman et al. 2000; Miller, Goldman, and others 2004), in the context of work on control of autonomous aerial vehicles.

```
(:method (get-airborne ?a)
 already-airborne ; method name
 ;; method precondition
 ((airborne ?a))
 ;; task network
 () ; no-op
 ;; alternative method
 rotorcraft-takeoff
 ;; precondition
 ((helicopter ?a))
 ;; task network
 ((vertical-takeoff ?a))
 ;; third alternative
 fixed-wing-takeoff
 ((fixed-wing ?a) ;prec
  (runway-segment ?a ?s ?d))
 ;; task network
 (:ordered (get-to-segment ?a ?s)
           (!takeoff ?a ?s ?d)))
```

Figure 1: Sample SHOP2 method definition.

Like other HTN planners, and unlike first-principles planners, SHOP2 searches top-down from a task or set of tasks, rather than chaining together primitive actions. SHOP2 and other HTN planners *decompose* complex tasks into more primitive sub tasks, building a plan tree, which terminates at leaves that correspond to primitive actions. SHOP2 uses PDDL actions (permitting conditional effects and quantification) as its primitives.[3]

In addition to actions, SHOP2's language provides *methods* for performing complex tasks. A method definition associates a task with a set of preconditions and a task network. When the preconditions are satisfied (more about this later), a task that matches the task in the method definition can be decomposed to the given task network. Figure 1 shows a very simple method indicating that there are three alternative ways of performing the task of getting an aircraft airborne: one for aircraft already airborne (no-op); one for a fixed-wing aircraft; and one for a helicopter.[4] Task networks are lists of tasks that may be constrained to be `:ordered`, or that can be executed in any order (`:unordered`). Method preconditions may contain disjunctions and quantification. Like other HTN planners, SHOP2 offers protections.

SHOP2 is *unlike* other HTN planners in its way of updating the state. In the way it applies actions, SHOP2 is a forward state-exploring planner. Whenever reducing a task

[2] `http://sourceforge.net/projects/shop`

[3] The original version of SHOP2 had its own primitive actions, but we have modified it to accept PDDL actions.

[4] As an efficiency measure, alternatives in a single method definition have if-then-else semantics.

```
(1)  proc find-plans(State, Tasks, Protects)
(2)    if Tasks = ∅ then return ∅ fi
(3)    ; nondeterministically choose a task t that has no predecessors
(4)    t = chooseFromTasks
(5)    if primitive(t)
(6)      then
(7)           ; nondeterministically choose an action instance for t
(8)           < State', Tasks', Protects' >=
(9)               applyOp(t, State, Protects)
(10)     else (t is a complex task)
(11)          ; nondeterministically choose a method for t
(12)          < r, R' >= reduction(State, t)
(13)          < State', Protects' >= applyOp(r, State, Protects)
(14)          Tasks' = subst(R', t, Tasks)
(15)   fi
(16)   P = find-plans(State', Tasks', Protects')
(17)   return (o · P)
(18) end find-plans
```

Figure 2: Pseudocode for SHOP2 plan search

by applying a method, SHOP2 will decompose all the way down to a leftmost primitive action, and update the current state with the effects of that action. This can be seen in the use of the *State* variable in the SHOP2 algorithm of Figure 2; at all times SHOP2 has a notion of the current state. This sets it aside from purely top-down planners (see, for example, the UMCP algorithm (Erol, Hendler, and Nau 1994b)). The forward planning nature of SHOP2 makes it amenable to integration with complex external reasoners, that may be able to compute state progressions, but not invert state changes (Nau cites CAD/CAM programs as an example).[5]

We give pseudocode for the SHOP2 main routine in Figure 2. We present this as a nondeterministic algorithm, to conserve space. There are choice points at steps 4,9,12 and 13. Alternatives there must be explored through search. Attempts to apply an action (steps 9 and 13) or perform a reduction (12) could fail and trigger backtracking.

Previous approaches

Erol et al. offer a semantics for UMCP, an HTN planner that was a direct ancestor of SHOP2 (Erol, Hendler, and Nau 1994b; 1994a; Erol 1995). UMCP is more expressive than SHOP2 but, perhaps for this reason, has never been fully implemented.[6] Erol's semantics derives state transition semantics for task networks by structural induction, starting from primitive task networks. Non-primitive task networks (methods are pairs of tasks and task networks), with variable parameters and non-primitive sub-tasks are interpreted as sets of primitive task networks that satisfy the constraints on the non-primitive task net.

The structural induction in Erol et al.'s semantic definitions parallels the planning algorithm in a way that makes it easy and natural to show that the planning algorithm is sound and complete. However, it is not particularly helpful in answering the kind of questions that interest us here. In particular, in order to reason about the correctness properties of a particular method in the plan library, it seems we must reason about all ways an instance of that method might be composed into primitive task networks, and from there draw conclusions about the state trajectories in the models for those composite task nets. This may well be possible, but the semantic scheme does not make it convenient.

Erol et al. pointed out that HTN planners are strictly more expressive than first principles planners. The reason is that an HTN method can constrain the state trajectory that is followed by a plan, rather than just its end state. Therefore, an HTN planner is capable of planning for goals that are not simple achievement goals, such as Davidson's example of running around a track three times, cited by McDermott (1982). McDermott points out the absurdity of characterizing this in terms of its end state of 'being in the same place, and tired.'

Our work is in many ways similar to that of Stephan and Biundo (1993; 1996). Like them, we are more interested in reasoning about the correctness of planner methods than about the correctness of the planner itself. However, their approach assumes a very complex logical reasoner as the planner, and the use of a very expressive language to capture the HTN methods. We, on the other hand, take the conventional planning approach of assuming a very simple action language, and a highly optimized algorithm that exploits the characteristics of the action language.

Work on the Golog procedural extension to the situation calculus is similar to Stephan and Biundo's work in attempting to use a very complex action representation for planning (Reiter 2001; Giacomo, Lespérance, and Levesque 2000). In the Golog framework, one attempts to find a valid instantiation of a Golog procedure (roughly corresponding to the decomposition of an HTN task network) by using search to explore nondeterministic choice points in the procedure. The situation calculus, with its solution to the frame problem, provides a means to project the effects of actions in this search process. Unlike more conventional planning work (like ours), this approach assumes that the procedures will be written so that this projection can be done efficiently, since the problem is so general as to resist more general solutions. Again, our work is more in the vein of mainstream planning, in assuming more restricted representations so that our algorithms can exploit them.

The contrast between the Golog approach and ours can very clearly be seen in the contrast between Gabaldon's work on the semantics of HTNs (2002) and our own. Gabaldon aims to show that the Golog approach can subsume HTN planning, which he does by providing translations of HTNs into Golog. He ends up with a very expressive framework, that uses the full power of Golog. We, on the other hand, wish to maintain the limitations of conventional HTNs that allow them to be computationally efficient, and to use the expressive power of the situation calculus and Golog only for the purposes of analysis and verification. Similar to Gabaldon's work is that of Baral and Son (1999). Baral and Son translate some HTN constructs (notably partial-ordering

[5] Dana Nau, personal communication.

[6] UMCP permits arbitrary inter-step constraints within a partially-ordered task network. SHOP2 has only limited partial-ordering and supports only limited protections.

and UMCP-style step naming) into Congolog. However, the result of their work is an *extension* of Congolog, with more expressive power; we use Congolog to study SHOP2, so we provide a *sublanguage* of Congolog, with strictly restricted expressive power, to facilitate analyses. McIlraith and her colleagues have done work that is almost the inverse of ours: they use Congolog as a comfortable, expressive notation, and compile it down to less expressive action theories for use by planners (Fritz, Baier, and McIlraith 2008).

Marthi et al. (2007) propose an HTN method semantics based on angelic nondeterminism alone — whereby the meaning of a method lies in the transitions made by successful plans containing those methods. They dismiss adversarial nondeterminism, because they are only concerned with the plan that is the product of planning. We are concerned with both angelic and adversarial nondeterminism: the former when generating plans, the latter when executing them. Even during planning, there is adversarial nondeterminism since unfavorable interactions between multiple goals can arise. In this paper, we exploit the expressive power of the Congolog framework to address these interactions.

Kambhampati et al. (1998) discuss the meaning of HTN methods while developing an algorithm to merge HTN and first-principles planning. Their work focuses on different issues than ours — less on execution and on analyzing the effect of interval constraints on execution and search. However, their notions of schema completeness and user intent are of great importance, and we hope to explore them in future work. Furthermore, integrating first principles planning, especially for replanning, is of great interest to us.

Modal temporal logics such as LTL and CTL have been used to characterize plan trajectories, both in order to encode heuristics about what would be a good plan (Bacchus and Kabanza 2000), and to encode more expressive goal specifications (Bacchus and Kabanza 1996; Gerevini and Long 2005). While the design of these logics capture important insights into how to capture state trajectory constraints, they are not convenient for reasoning about methods because they are *endogenous*; i.e., they do not make it convenient to talk about programs (or plans), because a particular program is implicit, rather than programs and their components being explicit entities to be reasoned about in the logic.[7]

HTN Semantics

Our semantic approach builds upon the semantics for concurrent situation calculus programs developed for ConGolog (Giacomo, Lespérance, and Levesque 2000).

ConGolog is a concurrent programming language for programs that interact with their environment. The semantics of ConGolog's primitive actions come from the situation calculus (Reiter 2001). That is, these actions are characterized by precondition and effect axioms, and a closure assumption is made to solve the frame problem. The critical components of the situation calculus theory of actions are:

Action precondition axioms, $\mathrm{Poss}(a, s)$**:** Action a may be executed in situation s.

Successor state function: $do(a, s)$ is the state that results from performing action a in state s.

Effect axioms: State conditions under which an action will cause a fluent to change. E.g., $has\text{-}fuel(a, s) \supset in(a, r, do(fly(a, r', r), s))$.

Successor state axioms: We assume that the set of actions is complete, and based on this assumption, generate a set of successor state axioms that captures all possible ways a fluent could come to (not) hold in a state after performing some action.

The actions in our SHOP2 methods are ADL-style PDDL actions, with conditional effects.[8] We can easily model these with the situation calculus, extracting the executability conditions ($\mathrm{Poss}(a, s)$ for actions a) from the preconditions. For example,

$$\mathrm{Poss}(\mathrm{takeoff}(ac, ap, r)) \equiv \mathrm{landed}(ac, ap) \wedge \mathrm{region}(ap, r)$$

We may also compile successor state axioms from a set of PDDL action definitions. E.g.,

$$\begin{aligned} \mathrm{in}(ac, r, do(a, s)) \quad \equiv \quad & \exists ap.a = \mathrm{takeoff}(ac, ap, r) \wedge \\ & \mathrm{region}(ap, r) \vee \\ & \exists r'.a = \mathrm{fly}(ac, r', r) \vee \\ & [\ in(ac, r) \vee \neg\exists r'.a = \mathrm{fly}(ac, r, r') \\ & \quad \vee\neg\exists ap.a = \mathrm{land}(ac, ap, r)\ ] \end{aligned}$$

This gives a solution to the frame problem, formalizing an extended STRIPS assumption that corresponds to the PDDL semantics, which we will need for our correctness proofs.

The constructs we need to give semantics for SHOP2 HTN methods are simple atomic actions, which we get from the situation calculus (but see below), concurrent composition ($\|$), sequential composition ($;$), which we take from the ConGolog semantics.[9] Later, we will augment these constructs with additional ones needed to capture HTN methods. ConGolog gives semantics to programs composed out of situation calculus using these constructs by means of the $\mathrm{Trans}(\delta, s, \delta', s')$ and $\mathrm{Final}(\delta, s)$ relations. Intuitively, $\mathrm{Trans}(\delta, s, \delta', s')$ means that a program, δ, in a situation, s, can evolve in one step to a new program δ' in a new situation, s'. $\mathrm{Final}(\delta, s)$ means that the program δ in state s, represents a terminating computation. Then the Do relation for a complex program, may be captured in terms of the transitive closure of Trans, Trans^*, and Final:

$$Do(\delta, s, s') \equiv \exists\delta'.\ \mathrm{Trans}^*(\delta, s, \delta', s') \wedge \mathrm{Final}(\delta', s') \quad (1)$$

Note that Do, is a relation, unlike do, since our programs may be nondeterministic. Also note that, while the original ConGolog semantics involve second order quantification, ours need not, because we have less expressive power

[7] See Kozen and Tiuryn (1990) for a discussion of endogeny in logics for reasoning about programs.

[8] For the moment, we ignore numerical fluents, and we do not permit truly simultaneous actions, as PDDL 2.1 does.

[9] For the purposes of this short paper, we omit the nondeterministic choice of value ($\pi\mu.\delta$) construct. In applications where we make use of the lifted nature of SHOP2, this is necessary to complete the semantic treatment. However, it is not an obstacle, and complicates the discussion unnecessarily here.

$$
\begin{aligned}
\mathrm{Trans}(nil, s, \delta', s') &\equiv \bot && (1)\\
\mathrm{Trans}(a, s, nil, s') &\equiv \mathrm{Poss}(a, s) \wedge s' = do(a, s) && (2)\\
\mathrm{Trans}(\delta_1;\ \delta_2, s, \delta', s') &\equiv [\exists \gamma.\, \mathrm{Trans}(\delta_1, s, \gamma, s') \wedge \delta' = \gamma;\ \delta_2] \vee [\mathrm{Final}(\delta_1, s) \wedge \mathrm{Trans}(\delta_2, s, \delta', s')] && (3)\\
\mathrm{Trans}(\delta_1 \parallel \delta_2, s, \delta', s') &\equiv [\exists \gamma.\, \mathrm{Trans}(\delta_1, s, \gamma, s') \wedge \delta' = \gamma \parallel \delta_2] \vee [\exists \gamma.\, \mathrm{Trans}(\delta_2, s, \gamma, s') \wedge \delta' = \delta_1 \parallel \gamma] && (4)\\
\mathrm{Final}(nil, s) &\equiv \top && (5)\\
\mathrm{Final}(a, s) &\equiv \bot && (6)\\
\mathrm{Final}(\delta_1;\ \delta_2, s) &\equiv \mathrm{Final}(\delta_1, s) \wedge \mathrm{Final}(\delta_2, s) && (7)\\
\mathrm{Final}(\delta_1 \parallel \delta_2, s) &\equiv \mathrm{Final}(\delta_1, s) \wedge \mathrm{Final}(\delta_2, s) && (8)\\
& && (9)
\end{aligned}
$$

Figure 3: Trans and Final definitions.

than full ConGolog. The Trans and Final definitions we will adopt from ConGolog are given in Figure 3.

Now we need a semantics for SHOP2 HTN methods. For this purpose, we give two axioms and an axiom schema that describes a set of axioms which capture what happens when expanding a method. This axiom schema will allow us to avoid second-order quantification, as used in the ConGolog semantics (Giacomo, Lespérance, and Levesque 2000).

$$
\begin{aligned}
\mathrm{Trans}(task(\vec{v}), s, \delta', s') \equiv\ & \exists \delta.\, \mathrm{Exp}^*(task(\vec{v}), s, \delta) \wedge \\
& \mathrm{Trans}(\delta, s, \delta', s')
\end{aligned} \quad (2)
$$

$$
\begin{aligned}
\mathrm{Exp}^*(task(\vec{v}), s, \delta) \equiv\ & \exists \delta'.\, \mathrm{Exp}(task(\vec{v}), s, \delta') \wedge \\
& [\mathrm{Exp}^*(\delta', s, \delta) \vee \\
& \delta = \delta']
\end{aligned} \quad (3)
$$

For Exp, instead of an axiom, we have an axiom *schema*:

$$
\begin{aligned}
&\mathrm{Exp}(task(\vec{v}), s, \delta') \equiv \\
&\quad \bigvee_{\mathcal{M}} Meth(task(\vec{v}), \mathcal{M}, \psi) \wedge \\
&\qquad Prec(\mathcal{M}, \phi) \wedge \\
&\qquad \exists \psi'.\, [\, \psi' \supseteq \psi \wedge \phi / \psi' [\![s]\!] \wedge \\
&\qquad\quad Body(\mathcal{M}, \delta) \wedge \delta' = \delta / \psi'\,]
\end{aligned} \quad (4)
$$

The above schema will expand into an axiom for each task expression, that is a disjunction over the different method definitions ($\mathcal{M}$). This provides us with an explicit closure over the set of method definitions, akin to the closure in our successor state axioms. Without this closure property we would not be able to draw conclusions about what *must* happen, only what *may* happen. Every method definition, $\mathcal{M} = \langle Prec(\mathcal{M}), Body(\mathcal{M}) \rangle$, where $Prec$ is the precondition component of the method definition, and $Body$ is its body (a task network). The $Meth$ relation maps between a task, a matching method definition, and a most general unifier (MGU): $Meth : task(\vec{v}) \longrightarrow \langle \mathcal{M}, \psi \rangle$. $task(\vec{v})$ is a task with a set of bound parameters. ψ' is a MGU that extends ψ such that the precondition expression, ϕ, is satisfied in the current state, s.

Note that our definition makes the testing of a method's preconditions an atomic operation together with the expansion of the method into its body. This must be done so that concurrent tasks do not get inserted between the testing of the preconditions and the method expansion. That is why we do not simply translate the method definition using Golog's test construct. The second feature is that we force the execution of at least one primitive action in a method body's (recursive) expansion. Again, this is done so that the preconditions of a method cannot be clobbered before the method's operation has begun.

We have not yet said how the preconditions ($Prec(\mathcal{M})$) and body of a method ($Body(\mathcal{M})$) are to be translated. The preconditions are simplicity itself: in SHOP2 the precondition expression is a formula, and is carried over unchanged.[10] The body is only a little more complicated. A SHOP2 task network is a set of tasks that can be grouped in `:ordered` and `:unordered` constructs. A group of ordered task networks, (`:ordered` $t_1 t_2 \ldots t_n$) translates into a sequential composition, $t_1;\ t_2 \ldots;\ t_n$, where t_i are translated recursively. Similarly, a group of unordered task networks (`:unordered` $t_1 t_2 \ldots t_n$) translates into a concurrent composition $t_1 \parallel t_2 \ldots \parallel t_n$ (again t_i must themselves be translated).

The last construct that we have left untranslated is the protection. We present a slight idealization of the mechanism used in the actual SHOP2 implementation. We allow any task (primitive action or method invocation) to be followed by the addition or deletion of protections; let us note this operation as $protect(\lambda)$, for some literal λ. We will also add an operation that removes such protections, $unprotect(\lambda)$. It is tempting to simply use ConGolog's sequential composition for this purpose, so that to carry out an action, a, and protect some literal, p would be translated as $a;\ protect(p)$. This is not appropriate, since in the presence of concurrently-executing tasks, a protection-clobbering action could slip in between a step that establishes a protected fluent, and the establishment of the protection. Consider the case of a method, M_1, in which a UAV is to put its sensor into state S_1, protect S_1, and do the observe(x) action, requiring S_1. M_1 is un-

[10] A subtlety is that variables in the preconditions are scoped over the body, as well.

ordered wrt M_2, in which the same UAV is to put its sensor into S_2, protect S_2, and do the $\mathrm{observe}(y)$ action. Without the atomic addition of the protection, we could see: UAV puts sensor into state S_1 (M_1); UAV puts sensor into state S_2 (M_2); planner attempts to protect S_1 [sic]. This signals the need for a special construction that will insure that the addition of protections can be bundled into an atomic transition with the execution of a primitive action.

We will add to our notation a new program connective, &, and allow program expressions of the form

$$a\& \left[protect(\lambda) \mid unprotect(\lambda)\right]^+$$

an action composed with one or more (un)protections. Now we must extend the Trans relationship to add protections to the arguments. This may most easily be done by encapsulating the original Trans definition as Trans'. Now we may define the & connective using the following relationships:

$$\begin{aligned} \mathrm{Trans}(\delta, p, s, \delta', p, s') &\equiv \mathrm{Trans'}(\delta, s, \delta', s') \quad (5) \\ &\quad \wedge p[\![s']\!] \\ \mathrm{Trans}(\delta \& \boldsymbol{Ps}, p, s, \delta', p', s') &\equiv \mathrm{Trans'}(\delta, s, \delta', s') \quad (6) \\ &\quad \wedge \mathrm{update}(p, \boldsymbol{Ps}, p') \\ &\quad \wedge p'[\![s']\!] \end{aligned}$$

The update operation is implemented as one would expect; literals newly protected are added to p, and literals newly unprotected are removed from p, to give p'.[11] It is not sufficient to atomically add protections after performing an action, we must also be able to atomically remove protections before performing an action, so that a protected literal cannot be clobbered between the removal of the protection and its intended consumer. The necessary axiom is a simple variant of (6).

Proof of correspondence: sketch We can prove the correspondence between the SHOP2 algorithm and the Golog semantics by proving a one-to-one correspondence between the steps of the algorithm and the corresponding Trans clauses, and a similar correspondence between the termination conditions of the *find-plans* procedure of Figure 2. The parallel between the termination conditions is one-to-one because the δ in the Golog semantics parallels the tasks list in *find-plans*.[12] The parallel between Trans and the algorithm comes in three components: (I) the choice of a task to expand from the tasks in the SHOP2 algorithm (step 4) parallels clauses (3) and (4) in Figure 3; (II) method expansion in SHOP2 parallels the definition of Exp (4); (III) primitive action execution is the same.

[11] In SHOP2 and in the interpreter for our semantics is a slightly more complex scheme associating counts with each protection. This allows for nested protections of the same fluent, e.g. $p(P), p(P), u(P), u(P)$ where P is protected until the second $u(P)$.

[12] There are some minor complexities because of the way the Golog semantics leave *nil* "on the stack" necessitating clauses (7,8) in Figure 3.

Implementation We have implemented a trace generator for the semantics by modifying the Prolog-based Legolog interpreter of Levesque and Pagnucco (2000). For the HTN semantics in this section, we have modified the Legolog interpreter, which models deterministic plan execution, to be able to enumerate all traces for a given plan library.

Examples With the above semantics, we may explore properties of a plan library. For example, consider a plan library in which we have a method for photographing a particular map location by an autonomous agent. Given a simple problem with a single photo target (and a completely connected map), we may demonstrate that this method is guaranteed to achieve its goal. However, we may additionally prove that, given two or more photo targets, the same method may generate arbitrarily bad plans if it interleaves working on these two goals — it move towards one, then towards the other, then back towards the first, etc.

We can demonstrate the flaw by taking the above interpreter and extending it to enable the interjection of additional actions into the plans. Doing this is a conservative over-approximation of what could happen when execution of our method is interleaved with arbitrary other methods. This method will (eventually) find any problem arising from such an interleaving, but it can find some problems that do not correspond to any actual interleaving. Indeed, when we allow the interpreter to put in additional movement actions, we see that our initial draft method can yield plans that violate an invariant we would like to see — that during the approach to the photo target, the vehicle should always be getting closer.

This would suggest adding protections that would preserve the position of the vehicle during its transit towards one target from interference by other motion goals.[13] Indeed, when we protect the position of the vehicle as it moves, the interpreter shows that other movements cannot be placed into the plan while the vehicle is approaching its target. Note that the protection approach suggested here is more flexible than simply forcing the rover to work on one goal at a time. What the protections do is hold control of the position of the rover. They leave the rover free to interleave activities that do not change its position, e.g., do uplink activities while moving towards the first target.

Semantics and execution

For the purposes of planning, nondeterminism is properly modeled as angelic nondeterminism. The agent can freely search for choices that meet its desires, because it is not executing any actions that change the world, it is only *projecting* the effects of those actions. These are the semantics that we have assumed up to this point. However, when considering execution of plans computed in advance of execution,

[13] Note that sketching out this example also suggests enhancing the expressive power of protections. Within the same overall framework, it would make the modeling easier if resources and properties (such as the position of a vehicle) could be protected, rather than being limited to protecting ground propositions.

choices will be fixed (at least for "classical" plans), and effects cannot, in general, be undone. Furthermore, we must take into account the possibility that the environment will evolve in ways beyond our control. In this section we discuss how to accommodate this in our semantics.

When analyzing the execution of a plan, we must take the plan produced off-line as our starting point. Following on the earlier proof of equivalence, we may either take the output of the SHOP2 planning process, and translate it into the Golog dialect, or we may reason about plan generation using the Golog semantics directly. Which is appropriate will depend on the task at hand. When reasoning about general cases (for example, all possible plans containing a given method) it will typically be easier to start with the Golog semantics.

Two parts of the earlier Golog semantics provide for angelic nondeterminism. The first is the interpretation of choice. This is limited to method choice up to now, and does not come into play in reasoning about execution, since all method choices will already have been made. The second is the definition of Do^* in terms of Trans and Final (1). This definition is inappropriate for use in analyzing plan execution because the use of Final removes from consideration any unsuccessful execution sequences. Consider how the plan-time semantics would treat a sequence of primitive actions that was ill-formed, in the sense that some action's preconditions will not be satisfied. The plan-time semantics would treat this sequence as not corresponding to any sequence of situations, because it cannot be completed. This is appropriate for plan-time, but clearly wrong for execution. When modeling execution the semantics should be a trace terminating immediately before execution of an action whose preconditions are not satisfied. When considering execution, we will wish to use not just Do, but also use Trans^* directly.

We may now introduce into this framework some consideration for exogenous actions or disturbances. To analyze the robustness of our plans, we may introduce a "demonic coprocess" that can introduce exogenous events into the planned action sequence:

$$\mathrm{Trans}_e(\delta, s, \delta', s') \equiv \mathrm{Trans}(\delta, s, \delta', s') \vee \mathrm{Exog}(s, s') \wedge \delta' = \delta \quad (7)$$

When engineering autonomous systems in dynamic domains, we want to develop systems that can handle some set of disturbances (exogenous actions) that are anticipated, but that cannot be directly controlled. These disturbances must be detected and handled (or "rejected," to use control theory terminology). We adopt the conventional approach of having a reactive executive that will execute projective plans while rejecting disturbances.

A problem in designing such systems is that it is difficult to know what disturbances need to be rejected, and what conditions need to be monitored by the executive. Our semantics helps answer this question. We augment the existing planning constructs by allowing the planner to add *monitors* to its plans. A monitor is similar to a protection, and acts the same way at plan time. Monitors are distinguished from protections in being carried over into execution time modeling and being modeled in the same way there. The intuition is that a monitor enables the executive to detect a disturbance that is a threat to successful execution *and to reject that disturbance*. We model this by modifying the Trans relation to eliminate execution of exogenous actions that are "blocked" by the presence of a monitor, in a way analogous to the way formula 6 blocks planned actions from clobbering protections.

Note that the monitors proposed here go beyond monitoring causal chains. We assume that the executive will take actions to repair the plan in case of precondition failure. However, in autonomous systems there are a number of reasons to have additional monitors. Monitors can be placed strategically to detect impending failures early, before causal failures. Related to this, monitors can be placed to trigger when a failure is only likely, not assured. Monitors can be placed to detect failures in trajectory properties that are not reflected in causal links.

We have extended the trace generator referred to in the semantics section to handle execution checking as well. The original Legolog interpreter (Levesque and Pagnucco 2000) models deterministic plan execution but, since it is an executive itself, it does not need the ability to generate multiple execution traces and so does not provide that capability. We have added that capability and (for planning-time reasoning) added the ability to model plan generation, with angelic nondeterminism.

Consider, for example, a rover application where the agent must traverse an area with known dust storms. The dust storms are modeled by an exogenous action of blocking the solar array. Assume that the solar array must be unblocked when drilling a rock (because of the current draw). A plan that clears the solar array, maneuvers the rover to the rock and begins drilling will successfully be generated by SHOP2, and satisfies our original plan semantics. However, in the presence of the dust storm demon, this plan is unsatisfactory, and it fails the check in our interpreter. But if we add a monitor to the method, the check succeeds. Further, the annotation on the plan acts as a requirement to the executive to be able to handle such a disturbance.

Remarks Kambhampati and Hendler (1992) propose a technique for replanning that exploits the protections of an HTN planner. This work inspired some of our thoughts on this work. However, as we were exploring the use of protections, we realized that the function of protections was not to protect against exogenous events. Protections aim to reduce the search space by ruling out interference *at plan time* between the planning agent's own actions. A plan library that has in it only protections that are aimed at plan-time search control will, in general, not provide sufficient annotations to derive monitors, and may provide annotations that are of no value. To return to our earlier example, if considering only search control, there would be no need for our rover agent to build in protections against it dropping dust on its own collector; even if it wanted to, it doesn't have that capability. On the other hand, it wouldn't need monitors against performing radio transmissions at inappropriate times — the

planning process will ensure that such activities are never generated. So while protections and monitors are similar in appearance, they serve importantly different purposes.

Conclusions

In this paper we have used the situation calculus and Golog to provide semantics for HTN planning. We have shown that this semantics sheds light on the meaning of the plans and methods of the HTN. The semantics also clarifies the meaning of protections and distinctions between planning-time and execution-time reasoning.

In future work, we intend to build upon this work in two directions. The first is to do more verification and validation of HTN plan libraries. Bienvenu *et al.* (2006) provide a translation of LTL progression in terms of the Golog Trans relationship. We can incorporate this together with our existing trace generator to perform model-checking style verification of HTN methods, both in terms of execution-time and plan-time properties, represented as LTL invariants. For the examples given in this paper, the invariants were only formulated informally, and checked by inspecting traces. In addition, the existing trace generator does not have the search control necessary to help find counterexamples efficiently. The second direction is to develop a new executive architecture for HTN plans that expands on the ideas presented here. This material focuses on the meaning of HTN plans, and what it would mean to build an executive for them. The next step is to bring a richer model of the executive into this framework, so that the constraints flow back and forth, with executive properties influencing planner properties and *vice versa*.

Acknowledgments

This article was supported by DARPA/IPTO and the Air Force Research Labodratory, Wright Labs under contract number FA8650-06-C-7606. This paper does not represent the official position or opinions of DARPA/IPTO or Air Force Research Labodratory, Wright Labs. Thanks to David Musliner for many helpful suggestions about the paper's structure. Thanks to John Maraist, Dana Nau and Ugur Kuter for discussions about HTNs and their semantics.

References

Bacchus, F., and Kabanza, F. 1996. Planning for temporally extended goals. In *Proceedings AAAI*, 1215–1222.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1–2):123–191.

Baral, C., and Son, T. C. 1999. Extending ConGolog to allow partial ordering. In *Agent Theories, Architectures, and Languages*, 188–204.

Bienvenu, M.; Fritz, C.; and McIlraith, S. 2006. Planning with qualitative temporal preferences. In Doherty, P.; Mylopoulos, J.; and Welty, C. A., eds., *KR2006*. Menlo Park, California: AAAI Press. 134–144.

Boddy, M. S.; Gohde, J.; et al. 2005. Course of action generation for cyber security using classical planning. In *ICAPS*, 12–21.

Currie, K., and Tate, A. 1991. O-Plan: the open planning architecture. *Artificial Intelligence* 52:49–86.

Erol, K.; Hendler, J.; and Nau, D. S. 1994a. HTN planning: Complexity and expressivity. In *Proceedings AAAI*, 1123–1128.

Erol, K.; Hendler, J.; and Nau, D. S. 1994b. UMCP: A sound and complete procedure for hierarchical task network planning. In *Proc. AIPS*, 249–254.

Erol, K. 1995. *Hierarchical task network planning :– formalization, analysis, and implementation*. Ph.D. Dissertation, University of Maryland at College Park. Available as UMCP technical report CS-TR-3624.

Fritz, C.; Baier, J. A.; and McIlraith, S. A. 2008. ConGolog, sin trans: Compiling ConGolog into basic action theories for planning and beyond. In *Proc. Conf. on Knowledge Representation and Reasoning*, 600–610.

Gabaldon, A. 2002. Programming hierarchical task networks in the situation calculus. In *AIPS'02 Workshop on On-line Planning and Scheduling*.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Technical Report RT 2005-08-47, Dept. of Electronics for Automation, University of Brescia, Brescia, Italy.

Giacomo, G. D.; Lespérance, Y.; and Levesque, H. J. 2000. ConGolog, A concurrent programming language based on situation calculus. *Artificial Intelligence* 121(1–2):109–169.

Goldman, R. P.; Haigh, K. Z.; Musliner, D. J.; et al. 2000. MACBeth: A multi-agent constraint-based planner. In *AAAI Workshop on Constraints and AI Planning*, 11–17.

Kambhampati, S., and Hendler, J. 1992. A validation structure-based theory of plan modification and reuse. *Artificial Intelligence* 55(2–3):193–258.

Kambhampati, S.; Mali, A. D.; and Srivastava, B. 1998. Hybrid planning for partially hierarchical domains. In *AAAI*, 882–888.

Kozen, D., and Tiuryn, J. 1990. Logics of programs. In *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. The MIT Press. 789–840.

Levesque, H. J., and Pagnucco, M. 2000. Legolog: Inexpensive experiments in cognitive robotics. In *Proceedings of the Second International Cognitive Robotics Workshop*.

Marthi, B.; Russell, S. J.; and Wolfe, J. 2007. Angelic semantics for high-level actions. In *Proc. ICAPS*, 232–239.

McDermott, D. V. 1982. A Temporal logic for reasoning about processes and plans. *Cognitive Science* 6:101–155.

Miller, C. A.; Goldman, R. P.; et al. 2004. A playbook approach to variable autonomy control. In *American Helicopter Society 60th Annual Forum*, 2146–2157.

Nau, D.; Au, T. C.; et al. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Reiter, R. 2001. *Knowledge in Action*. MIT Press.

Stephan, W., and Biundo, S. 1993. A new logical framework for deductive planning. In *Proc. IJCAI*, 32–38.

Stephan, W., and Biundo, S. 1996. Deduction-based refinement planning. In *Proc. AIPS*, 213–220.

Wilkins, D. 1988. *Practical Planning*. Morgan Kaufmann.