Angelic Semantics for High-Level Actions

Bhaskara Marthi
CSAIL
Massachusetts Insitute of Technology
Cambridge, MA 02139
bhaskara@csail.mit.edu

Stuart Russell
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
russell@cs.berkeley.edu

Jason Wolfe *
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
jawolfe@cs.berkeley.edu

Abstract

High-level actions (HLAs) lie at the heart of hierarchical planning. Typically, an HLA admits multiple refinements into primitive action sequences. Correct descriptions of the effects of HLAs may be essential to their effective use, yet the literature is mostly silent. We propose an *angelic semantics* for HLAs, the key concept of which is the set of states reachable by some refinement of a high-level plan, representing uncertainty that will ultimately be resolved in the planning agent's own best interest. We describe upper and lower approximations to these reachable sets, and show that the resulting definition of a high-level solution automatically satisfies the upward and downward refinement properties. We define a STRIPS-like notation for such descriptions. A sound and complete hierarchical planning algorithm is given and its computational benefits are demonstrated.

Introduction

Since the early days of AI (Simon 1962), hierarchical structure in behavior has been recognized as perhaps the most important tool for coping with complex environments and long decision horizons. Humans, who execute on the order of one trillion primitive motor commands in a lifetime, appear to make heavy use of hierarchy in their decision making—we entertain and commit to (or discard) high-level actions such as "write an ICAPS paper" and "run for President" without preplanning the motor commands involved, even though these high-level actions must eventually be refined down to motor commands (through many intermediate levels of hierarchy) in order to take effect.

For the purposes of this paper, hierarchical structure is supplied in the form of a library of *high-level actions* (or HLAs), in addition to the primitive action descriptions used in standard planning. Each HLA admits one or more *refinements* into *sequences* of actions, which may include other HLAs. (More complex definitions of hierarchy are possible, of course, but this suffices for now.)

The principal computational benefit of using high-level actions seems obvious: high-level plans are much shorter, and ought to be much easier to find. This benefit can be realized only if high-level solutions can be identified—that is, only if we can establish that a given high-level plan has a primitive refinement that achieves the goal. (This is the motivation for the much-sought-after *downward refinement property* (Bacchus & Yang 1991), which requires that every high-level solution must be refinable into a primitive solution.) Another important benefit is that a situated agent may begin executing primitive actions corresponding to the initial step of a high-level solution without having first reduced the remaining steps to primitive actions.

For this approach to work, the planner needs to know what high-level actions *do*. Yet even though many hierarchical planners use precondition–effect annotations for HLAs, their exact meaning is often unclear. After the collapse of the "hierarchical" track at the first Planning Competition, McDermott (2000) wrote as follows:

> *The semantics of hierarchical planning have never been clarified . . . the hierarchical planning community is used to thinking of library plans as advice structures, which was a drastic departure from our assumption that the basic content of the plan library contained no "advice", only "physics". . . . The problem is that no one has ever figured out how to reconcile the semantics of hierarchical plans with the semantics of primitive actions.*

Our analysis of the literature, sketched briefly in the "Related Work" section towards the end of the paper, suggests this is slightly pessimistic but essentially correct.

Our aim in this paper is to rectify this situation and thereby realize the full benefits of hierarchical structure in planning. Our approach is a simple one: we provide descriptions of the effects of HLAs that are *true*—that is, they follow logically from the refinement hierarchy and the descriptions of the primitive actions. If achievement of the goal is entailed by the true descriptions of a sequence of HLAs, then that sequence must, by definition, be reducible to a primitive solution. Conversely, if the sequence provably fails to achieve the goal, it is not reducible to a primitive solution. Thus, the downward refinement property and its converse are automatically satisfied.[1] Finally, if the descriptions entail neither success nor failure of a given plan, then further refinement will resolve the uncertainty.

So far, so good. But what can be truly asserted about high-level actions? The first such assertions were the

*The authors appear in alphabetical order.

[1] Bacchus and Yang (1991) assert that it is "naive" to expect downward refinement to hold in general; but downward refinement can fail only if HLAs are allowed to have *false* descriptions. Such descriptions are common in ABSTRIPS hierarchies.

precondition–effect descriptions of *macro-operators* in the early STRIPS system (Fikes & Nilsson 1971); these descriptions were true, and indeed were derived automatically from solutions found by the planner. Each STRIPS macro-operator, however, had exactly one refinement, whereas the tricky issues arise with multiple refinements. The most common proposal is that the effects of an HLA are those that occur in *all* of its refinements (Tate 1977; Yang 1990; Russell & Norvig 2003). This is equivalent to the way in which nondeterminism is handled in planning: a plan works iff it works no matter how the choices are made. But this treatment ignores the fact that *the planning agent itself* (not an adversary) chooses which refinement to make. Instead of requiring effects to hold under *all* refinements, then, we are interested in what happens under *some* refinement.

A similar distinction between adversarial and nonadversarial refinements of abstract specifications occurs in the semantics of programming languages. For example, a multithreaded program is correct only if it works under *all* legal thread sequencings, because the actual sequencing depends on unknown ("adversarial") timing properties of the machine and process scheduler. On the other hand, a nondeterministic algorithm succeeds iff *some* sequence of choices succeeds—the choices are assumed to be made with the aim of succeeding. In the programming languages literature, the term *angelic nondeterminism* is used for this case (Jagadeesan, Shanbhogue, & Saraswat 1992). Thus, it seems reasonable to call our proposed semantics an *angelic semantics* for HLAs.

To describe the angelic semantics correctly, we begin with the concept of a *reachable state*: a state s' is reachable from s by a given HLA a iff *some* primitive refinement of a takes s into s'. The *exact description* of an HLA is simply its reachable set, specified as a function of s. If an HLA's reachable set from the start state intersects the goal, then the HLA is a solution. We can extend this notion in an obvious way to sequences of HLAs, and thereby derive a hierarchical planning algorithm with the desirable properties mentioned above.

To illustrate the basic idea, let us consider a domain where the state is described by propositional fluents u, v, and w and where there is an HLA a that has two possible refinements r_1 and r_2 into primitive sequences. The sequence induced by r_1 has effects $+u+v$, while r_2 has effects $+u-v$. (Both leave w unchanged.) Under adversarial semantics, we can be sure that a achieves u, but we can say nothing about v. In particular, under adversarial semantics, a is not a solution to the problem of achieving $u \wedge v$. Yet it is obvious that a *is* a solution; we simply have to choose the refinement r_1. Under angelic semantics, a always achieves u and possibly achieves v or $\neg v$ (agent's choice). Ordinary STRIPS notation cannot capture this—note that leaving v out of the effect list implies that v is always unchanged—so we introduce an extension called NCSTRIPS (nondeterministic conditional STRIPS). In this notation, the exact description of the effect of a is $+u \pm v$, where $\pm$ denotes that, by choosing the appropriate refinement, we can either add or delete the corresponding proposition.

It turns out, however, that even with this notational extension, exact descriptions of reachable sets are seldom concise enough to be usable. Instead, we introduce both *complete descriptions*, which provide an upper bound (superset) on the reachable set, and *sound descriptions*, which provide a lower bound (subset) on the reachable set. When extended from actions to sequences, complete descriptions support proofs that a sequence *cannot* reach the goal under *any* refinement, while sound descriptions support proofs that a sequence *surely* reaches the goal under *some* refinement.

Given a language for expressing sound and complete descriptions, it is conceptually straightforward to derive algorithms for constructing them automatically, starting from the primitive actions (which always have exact descriptions in the standard setting) and working up the hierarchy. We defer this issue to a future paper, pointing out for now that there are tradeoffs among the expressiveness of the language and the exactness and conciseness of the descriptions. We illustrate these tradeoffs in an example domain—the *warehouse world*, which is a variant of the blocks world. For our NCSTRIPS language, we find that nonempty sound descriptions are sometimes hard to come by. We introduce a third type of description called *sound-intersecting*, which has many of the useful properties of sound descriptions but—at least in our limited experience—seems a better fit to real domains.

Based on this approach to describing HLAs, we describe a simple, provably sound and complete hierarchical planning algorithm that performs a "top-down" forward search. We report on preliminary experiments in the warehouse world, demonstrating order-of-magnitude, complementary speedups from using sound and complete descriptions.

The representations and algorithms described in the body of the paper are just one way in which the overall approach can be fleshed out; many other ways are possible. There is also a lot more work to do on the methods we have chosen. These issues are discussed in the final section of the paper.

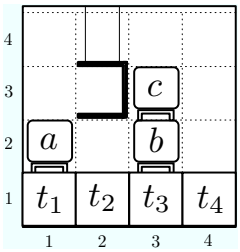
Hierarchical Planning

Planning Problems

Since the main semantic issues regarding HLAs are orthogonal to the particular representation used for states and actions, we will define the concepts in a representation-independent way and illustrate them on a running example. For our purposes, a (deterministic, fully observable) *planning problem* consists of a set S of states, an initial state s_0, a set $\mathcal{G} \subseteq S$ of goal states, a set $\mathcal{L}$ of *primitive actions*, and a transition function $f : S \times \mathcal{L} \rightarrow S$. The meaning of this function is that doing a in s leads to state $f(s,\ a)$. For simplicity, we omit strict action preconditions, assuming instead that actions leave the state unchanged if their preconditions are not met. The transition function can be overloaded to take in sequences of primitive actions; if $\mathbf{a} = (a_1, \ldots, a_m)$, then $f(s,\ \mathbf{a})$ is the result of doing the actions $a_1, \ldots, a_m$ in order, starting at s. The planning problem is to find $\mathbf{a} \in \mathcal{L}^*$ such that $f(s_0,\ \mathbf{a}) \in \mathcal{G}$.[2]

A popular representational choice is to make the state set S be the set of truth assignments to some set of ground propositions, describe the goal using a logical formula, and describe the transition function using the STRIPS language (Fikes & Nilsson 1971).

[2]We restrict our attention to totally ordered planning, and do not consider finding optimal (e.g., shortest) plans.



predicate	interpretation
R(x_i, x_j)	x_i one square right of x_j
U(y_i, y_j)	y_i one square up from y_j
POS(x, y)	gripper at (x, y)
FACINGR	gripper facing right
HAVE(b)	gripper holding block b
EMPTY	gripper holding nothing
AT(b, x, y)	block b at (x, y)
FREE(x, y)	no block at (x, y)
ON(b, c)	block b on c
CLEAR(c)	no block on c

Figure 1: Left: An example instance of a 4x4 warehouse world planning problem. The goal is to have ON(c, t_2) $\wedge$ ON(a, c). Right: predicates used to encode the domain. R and U are constant, and all other predicates are fluents.

TurnR():	*also* TurnL()
pre:	POS(x, y) $\wedge$ $y = h$
eff:	+FACINGR
Right():	*also* Left(), Up(), Down()
pre:	POS(x_s, y_s) $\wedge$ R(x_t, x_s) $\wedge$ FREE(x_t, y_s)
eff:	−POS(x_s,y_s), +POS(x_t,y_s)
GetR():	*also* GetL()
pre:	POS(x_g, y) $\wedge$ EMPTY $\wedge$ FACINGR $\wedge$ R(x_b, x_g) $\wedge$ AT(b, x_b, y) $\wedge$ CLEAR(b) $\wedge$ ON(b, c)
eff:	−ON(b, c), −AT(b, x_b, y), −EMPTY, +CLEAR(c), +FREE(x_b, y), +HAVE(b)
PutR():	*also* PutL()
pre:	POS(x_g, y_{bg}) $\wedge$ HAVE(b) $\wedge$ FACINGR $\wedge$ R(x_{bc}, x_g) $\wedge$ U(y_{bg}, y_c) $\wedge$ AT(c, x_{bc}, y_c) $\wedge$ CLEAR(c)
eff:	−CLEAR(c), −FREE(x_{bc}, y_{bg}), −HAVE(b), +ON(b, c), +AT(b, x_{bc}, y_{bg}), +EMPTY

Figure 2: STRIPS descriptions for $w \times h$ warehouse world domain.

To illustrate some of the issues, we use a simple propositional domain, a variant of the well-studied blocks world with discrete spatial constraints added. In this *warehouse world* (see Figure 1), blocks are stacked on a 1-d table of fixed length. A forklift-style gripper can pick up and stack blocks from the left or right side, as long as it is facing the target. The gripper can also move to adjacent free squares, and can turn (to face the other way) when in the top row. A partial STRIPS domain description is provided in Figure 2.[3] Problems in this domain can be quite challenging, due to constraints on the gripper position imposed by the ceiling and by the blocks themselves; for instance, the shortest plan that solves the example problem in Figure 1 is 50 steps long.

Hierarchy and Refinements

A *hierarchical planning problem* $\mathcal{P}$ consists of a planning problem together with an *action hierarchy* $\mathcal{H} = (\mathcal{A}, I, \mathcal{T})$. $\mathcal{A}$ is a set of *high-level actions*. Let $\tilde{\mathcal{A}} = \mathcal{A} \cup \mathcal{L}$ be the set of all actions, both high-level and primitive. For each $a \in \mathcal{A}$, there is a set $I(a)$ of possible *immediate refinements*, each of which is a finite sequence $(a_1, a_2, \ldots, a_m) \in \tilde{\mathcal{A}}^*$. Finally, $\mathcal{T} \subseteq \tilde{\mathcal{A}}$ is the set of actions that can be done at the top level. To *refine* a sequence $\mathbf{a} = (a_1, \ldots, a_m)$ is to replace one of the a_i by one of its immediate refinements. A sequence

[3]For ease of exposition we introduce new variables in preconditions when they are determined uniquely by the other arguments and state, rather than cluttering up the argument lists; it is trivial to translate this into correct PDDL definitions.

Nav(x_t, y_t)

pre	POS(x_s, y_s)
ref. 1	*pre:* $x_s{=}x_t \wedge y_s{=}y_t$ *ref:*
ref. 2	*pre:* R(x, x_s) $\wedge$ FREE(x, y_s) $\wedge$ $(x_s, y_s) \neq (x_t, y_t)$ *ref:* Right(), Nav(x_t, y_t)
ref. 3-5	Like refinement 2, but with Left(), Up(), or Down().
comp.	else → $\tilde{-}$POS(x_s, y_s), $(\forall xy)\tilde{+}$POS(x, y)
sound	POS(x_s, y_s) $\wedge$ FREE(x_t, y_t) $\wedge$ $(\forall\, x)$ FREE(x, h) →−POS(x_s, y_s), +POS(x_t, y_t) else → {}

Navigate(x_t, y_t)

pre	POS(x_s, y_s)
ref. 1	*pre:* *ref:* Nav(x_t, y_t)
ref. 2	*pre:* ¬FACINGR *ref:* Nav(x_s, h), TurnR(), Nav(x_t, y_t)
ref. 3	*pre:* FACINGR *ref:* Nav(x_s, h), TurnL(), Nav(x_t, y_t)
comp.	else → $\tilde{-}$POS(x_s, y_s), $\tilde{\pm}$FACINGR, $(\forall xy)\tilde{+}$POS(x, y)
sound	POS(x_s, y_s) $\wedge$ FREE(x_t, y_t) $\wedge$ $(\forall\, x)$ FREE(x, h) →−POS(x_s, y_s), +POS(x_t, y_t), $\tilde{\pm}$FACINGR else → {}

MoveBlock(b, c)

pre	CLEAR(b) $\wedge$ ON(b, a) $\wedge$ AT(b, x_b, y_b) $\wedge$ CLEAR(c) $\wedge$ AT(c, x_c, y_c) $\wedge$ U(y_t, y_c) $\wedge$ POS(x_g, y_g) $\wedge$ EMPTY
ref. 1	*pre:* R(x_b, x_i) $\wedge$ FREE(x_i, y_b) $\wedge$ R(x_c, x_j) $\wedge$ FREE(x_j, y_t) *ref:* Navigate(x_i, y_b), GetR(), Navigate(x_j, y_t), PutR()
ref. 2-4	Like refinement 1, but with GetL() and/or PutL().
comp.	else → $\tilde{-}$ON(b, a), $\tilde{+}$ON(b, c), $\tilde{-}$CLEAR(c), $\tilde{+}$CLEAR(a), $\tilde{-}$FREE(x_c, y_t), $\tilde{+}$FREE(x_b, y_b), $\tilde{-}$AT(b, x_b, y_b), $\tilde{+}$AT(b, x_c, y_t), $\tilde{-}$EMPTY,$\tilde{+}$HAVE(b), $\tilde{-}$POS(x_g, y_g), $(\forall xy)\tilde{+}$POS(x, y), $\tilde{\pm}$FACINGR
sound	else → {}

Figure 3: A hierarchy for the warehouse world domain. There are three HLAs, each of which has an overall precondition, a set of possible immediate refinements with preconditions, and sound and complete descriptions specified in the NCSTRIPS language.

$\mathbf{b}$ is a *refinement* of a sequence $\mathbf{a}$ iff $\mathbf{b}$ can be obtained by repeated refinements starting with $\mathbf{a}$, and a *primitive refinement* iff, in addition, it consists only of primitive actions. We define $R^*(\mathbf{a})$ as the set of all primitive refinements of $\mathbf{a}$. A sequence is *consistent with* $\mathcal{H}$ if it is a refinement of some sequence in $\mathcal{T}^*$. In hierarchical planning, we restrict attention to plans that are consistent with $\mathcal{H}$.[4]

Figure 3 shows a hierarchy for our running example, with various annotations that will be described in this and the following section. In this hierarchy, there are three HLAs: $\mathcal{A} = \{\mathsf{Nav}, \mathsf{Navigate}, \mathsf{MoveBlock}\}$. Nav($x$, y) attempts to navigate the gripper from its current position to (x, y). It has five possible immediate refinements: do nothing, or execute a primitive move step followed by a Nav. Navigate(x, y) attempts to navigate while possibly turning the gripper. It either directly Navs to the target, or Navs to the top row, turns, and then Navs to the target. Finally, MoveBlock(b, c) attempts to move a block b to on top of c. It has four immediate refinements, each of which Navigates to one side of b, picks it up, Navigates to one side of the target location, and puts b on c. The refinements differ only in the sides from

[4]Technically, to be consistent with the hierarchy, a primitive plan must also satisfy all HLA and refinement preconditions at points in the plan where their primitive refinements begin.

which the block is picked up and put down. While this hierarchy is simple, it is also quite general; all plans we care about can be generated solely by refinements of sequences of MoveBlock actions (i.e., $\mathcal{T} = \{\mathsf{MoveBlock}(\cdot)\}$). For instance, the example problem can be solved by moving c on a, b on t_4, c on t_3, a on b, c on t_2, and finally a on c.

High-Level Action Descriptions

As explained in the introduction, the set $R^*(a)$ of primitive refinements of HLA a may be large. A key concept in our approach is the set of states reachable by these sequences:

Definition 1. Given a hierarchical planning problem $\mathcal{P}$, the *exact reachable set* of a high-level action sequence $\mathbf{a}$ from state s is given by

$$F(s, \mathbf{a}) = \{s' : \exists \mathbf{b} \in R^*(\mathbf{a})\ f(s, \mathbf{b}) = s'\}$$

Remark. If $\mathbf{a}$ is primitive, $F(s, \mathbf{a}) = \{f(s, \mathbf{a})\}$.

Definition 2. An action sequence $\mathbf{a} \in \tilde{\mathcal{A}}^*$ *solves* hierarchical planning problem $\mathcal{P}$ iff $F(s_0, \mathbf{a}) \cap \mathcal{G} \neq \emptyset$.

In other words, a high-level plan is a solution iff the goal is reachable by some primitive refinement.

If we could compute $F(s, \mathbf{a})$ efficiently, during planning we would only need to refine action sequences known to solve the problem at hand. But this is not possible in general. (Proofs are deferred to the full paper due to lack of space.[5])

Theorem 1. *There exist hierarchical planning problems $\mathcal{P}_n, n \in \mathbb{N}$ for which the state descriptions, hierarchy size, and plan lengths are all polynomial in n and the transition model for primitive actions can be evaluated in polynomial time, such that, if there exists an algorithm that takes in $\mathcal{P}_n$ and a state s and decides in polynomial time whether the goal state of $\mathcal{P}_n$ is in $F(s, a)$ for some HLA a in $\mathcal{P}_n$, then P=NP.*

Thus, we consider instead principled *compact approximations* to F that still allow for exact inferences about what states a high-level plan can possibly or definitely reach. The framework we present is representation-independent; in order to illustrate and implement our definitions, however, we need a concrete way to specify sets of states and transition functions. In this paper, we use NCSTRIPS (Nondetermistic Conditional STRIPS), as illustrated in Figure 3. This language allows for a set of conditional effects with exhaustive, disjoint conditions,[6] each of which includes an add and delete list (as in the STRIPS language). In addition, effects may include *possible* adds and deletes, denoted by $\tilde{+}$, $\tilde{-}$, and $\tilde{\pm}$ (short for possible-add and -delete). A possibly-added proposition appearing in a complete (resp. sound) description of an action can possibly (resp. definitely) be made true by some refinement of that action.

Complete Descriptions

We first consider *complete descriptions*, which specify *supersets* of the exact reachable sets of HLAs. These allow us to prune a high-level plan that cannot possibly reach the goal, without explicitly considering its refinements. For example, consider the complete description of Navigate in Figure 3, which asserts that from any state, Navigate(x, y) may delete the current gripper position, add any other gripper position, and turn the gripper, but cannot affect any other propositions. This description allows us to infer that, e.g., no sequence of Navigate actions alone can solve the example problem, because Navigate cannot change the ON predicate.

Definition 3. A function $g_a : S \to 2^S$ is a *complete description* of an HLA a iff, for every $s \in S$, $g_a(s) \supseteq F(s, a)$.

We can extend a complete description g_a to a function $\bar{g}_a$ on sets of states, defined by $\bar{g}_a(\sigma) = \bigcup_{s \in \sigma} g_a(s)$. Then, we define the *complete reachable set* G of a high-level sequence as a composition of descriptions of its component actions:

Definition 4. Given complete descriptions g_a for all actions, the *complete reachable set* of $\mathbf{a} = (a_1, \ldots, a_N)$ from some set of states $\sigma \subseteq S$ is given by $G(\sigma, \mathbf{a}) = \bar{g}_{a_N} \circ \ldots \circ \bar{g}_{a_1}(\sigma)$.

The complete reachable set G can be used to support claims of the form "If a certain high-level sequence must lead to a state in σ, then following with high-level sequence $\mathbf{a}$ must lead to a state in $G(\sigma, \mathbf{a})$."

Theorem 2. *For all $s \in \sigma \subseteq S$, $\mathbf{a} \in \tilde{\mathcal{A}}^*$, and $\mathbf{b} \in R^*(\mathbf{a})$, $f(s, \mathbf{b}) \in G(\sigma, \mathbf{a})$.*

Corollary 1. *A plan $\mathbf{a} \in \tilde{\mathcal{A}}^*$ cannot possibly reach the goal (by any refinement) if $G(\{s_0\}, \mathbf{a}) \cap \mathcal{G} = \emptyset$.*

In summary, given a high-level plan, we can progress a set of possibly reachable states (starting with a set containing only the initial state) through the complete descriptions of the plan's component actions in sequence. If the final set does not intersect the goal, the high-level plan cannot solve the planning problem and need not be refined further.

Sound Descriptions

Sound descriptions are dual to complete ones, in that they specify *subsets* of the exact reachable sets of HLAs. For example, the sound description of Navigate(x, y) in Figure 3 asserts that if (x, y) and all squares in the top row are free, then Navigate can definitely move the gripper to (x, y), possibly turning it in the process; otherwise, no states are known to be achievable (denoted by the special $\{\}$ effect). Now, consider the problem of achieving ON(c, a) from the initial state in Figure 1. Using the sound description of Navigate (along with the primitive descriptions of GetL and PutL), we can conclude that the high-level plan Navigate$(4, 3)$, GetL, Navigate$(2, 3)$, PutL is guaranteed to reach this goal.

Definition 5. A function $h_a : S \to 2^S$ is a *sound description* of a iff, for every $s \in S$, $h_a(s) \subseteq F(s, a)$.

We extend h_a to a function $\bar{h}_a(\sigma) = \bigcup_{s \in \sigma} h_a(s)$ and then define the *sound reachable set* H as before:

Definition 6. Given sound descriptions h_a for all actions, the *sound reachable set* of $\mathbf{a} = (a_1, \ldots, a_N)$ from some set of states $\sigma \subseteq S$ is given by $H(\sigma, \mathbf{a}) = \bar{h}_{a_N} \circ \ldots \circ \bar{h}_{a_1}(\sigma)$.

The sound reachable set H can be used to support claims of the form "If we can reach any state in σ with a certain high-level sequence, then by following with high-level sequence $\mathbf{a}$, we can reach any state in $H(\sigma, \mathbf{a})$."

[5] Full paper, implementation, and test scripts available at `http://www.cs.berkeley.edu/~jawolfe/angelic/`

[6] In other words, we assume that exactly one effect applies in each state. For ease of exposition, we also assume that effect conditions are conjunctive (which can be enforced by preprocessing).

Theorem 3. *For all $\sigma \subseteq S$, $\mathbf{a} \in \tilde{\mathcal{A}}^*$, and $s' \in H(\sigma, \mathbf{a})$, there exists $\mathbf{b} \in R^*(\mathbf{a})$ and $s \in \sigma$ such that $f(s, \mathbf{b}) = s'$.*

Corollary 2. *A plan $\mathbf{a} \in \tilde{\mathcal{A}}^*$ can definitely reach the goal (by some refinement) if $H(\{s_0\}, \mathbf{a}) \cap \mathcal{G} \neq \emptyset$.*

Given a high-level plan, we can progress a set of definitely reachable states (starting with a set containing only the initial state) through the sound descriptions of the plan's component actions in sequence. If the final set intersects the goal, the plan can be refined to a primitive solution.

Moreover, if we want to actually find such a refinement, we can use the following theorem to decompose this task into separate subproblems, one for each HLA in the plan:

Theorem 4. *If for $\mathbf{a} = (a_1, \ldots, a_N)$ there exists $s_N \in H(\{s_0\}, \mathbf{a}) \cap \mathcal{G}$, a sequence $s_0, \ldots, s_N$ can be found by greedily choosing each s_{i-1} (starting with s_{N-1}) such that $s_{i-1} \in H(\{s_0\}, (a_1, \ldots, a_{i-1}))$ and $s_i \in h_{a_i}(s_{i-1})$. Moreover, for each a_i there exists $\mathbf{b}_i \in R^*(a_i)$ such that $f(s_{i-1}, \mathbf{b}_i) = s_i$. Concatenating any such $\mathbf{b}_i$ yields a primitive refinement $\mathbf{b} \in R^*(\mathbf{a})$ such that $f(s_0, \mathbf{b}) = s_N \in \mathcal{G}$.*

In the above example, the high-level Navigate(4, 3), GetL, Navigate(2, 3), PutL has only one soundly reachable state that satisfies the goal of having c on a. This state has the gripper at $(2,\ 3)$ facing left; we call it s_4. We then regress backwards from s_4, choosing s_3 in which the gripper is holding c, then s_2 in which the gripper is at $(4,\ 3)$ facing left, and finally s_1 in which c is still on b. s_0, of course, is just the initial state. Given these intermediate states, a primitive refinement of the plan can be found by solving separate planning problems, one for each Navigate HLA. Solving the first such problem of reaching s_1 from s_0 by a refinement of Navigate(4, 3) might yield the primitive sequence Up, Right, Right, Down, and solving the second such problem might yield Up, Left, Left, Down. Concatenating the results gives the primitive refinement Up, Right, Right, Down, GetL, Up, Left, Left, Down, PutL, which solves the original problem.

Sound-Intersecting Descriptions

In some domains, it is difficult to construct compact but useful sound descriptions. For instance, suppose we add to our domain a nuisance predicate describing the position of some gear in the gripper's mechanism, which is changed by each action in a complex but deterministic way based on the current state. With this modification, there is no longer a compact sound description of Navigate, since picking out concrete achievable states (which must specify the gear position) would require detailed conditioning on the source state.

However, we do know that Navigate can achieve the same values for the old propositions that it could before, and for each such combination of values, there is *some* possible gear position (but we don't want to say which one). In NCSTRIPS, we can represent this by adding to our previous sound description *intersecting*-possible-add and -delete lists containing all possible gear positions. The semantics is that we choose the direction the gripper faces, and then an adversary chooses the gear position. This compact description allows us to prove that the same plans as before succeed in this new domain. We call it a *sound-intersecting description*, because it describes a set of sets of states, such that we can *definitely* achieve *at least one* state from each set.

Definition 7. A function $j : S \to 2^S$ is an *intersecting description* of a iff for every s, $j(s) \cap F(s, a) \neq \emptyset$.

Definition 8. A *sound-intersecting description* of a is a set $k_a = \{j_i\}$ of intersecting descriptions of a.

We first extend the j_i to functions $\bar{j}_i(\sigma) = \bigcup_{s \in \sigma} j_i(s)$ on sets of states. Then, given a set of sets of states $\Sigma = \{\sigma_{i'}\}$, define $\bar{k}_a(\Sigma) = \{\bar{j}_i(\sigma_{i'})\}$ (ranging over all $\bar{j}_i$ and $\sigma_{i'}$).

Definition 9. Given sound-intersecting descriptions k_a for all actions, the *sound-intersecting reachable set of sets* of $\mathbf{a} = (a_1, \ldots, a_N)$ from some set of sets of states $\Sigma \subseteq 2^S$ is given by $K(\Sigma, \mathbf{a}) = \bar{k}_{a_N} \circ \ldots \circ \bar{k}_{a_1}(\Sigma)$.

A sound-intersecting reachable set K can be used to support claims of the form, "If we are able to reach some state in each set in Σ, then by following with high-level sequence $\mathbf{a}$, we can reach some state in each set in $K(\Sigma, \mathbf{a})$."

Theorem 5. *For all sets of sets of states $\Sigma \subseteq 2^S$, action sequences $\mathbf{a} \in \tilde{\mathcal{A}}^*$, and sets of states $\sigma' \in K(\Sigma, \mathbf{a})$, there exists a set $\sigma \in \Sigma$ such that, for any $s \in \sigma$, there is a primitive refinement $\mathbf{b}$ of $\mathbf{a}$ for which $f(s, \mathbf{b}) \in \sigma'$.*

Sound-intersecting descriptions, like sound ones, can be used to guarantee the existence of a successful primitive refinement of a given plan, and also to decompose the problem of finding such a primitive refinement:

Corollary 3. *A plan $\mathbf{a}$ can definitely reach the goal (by some refinement) if there exists $\sigma \in K(\{\{s_0\}\}, \mathbf{a})$ s.t. $\sigma \subseteq \mathcal{G}$.*

Theorem 6. *If for $\mathbf{a} = (a_1, \ldots, a_N)$ there exists $\sigma_N \in K(\{\{s_0\}\}, \mathbf{a})$ s.t. $\sigma_N \subseteq \mathcal{G}$, a sequence $\{s_0\}, \sigma_1, \ldots, \sigma_N$ can be found by greedily choosing each σ_{i-1} (starting with σ_{N-1}) so that $\sigma_{i-1} \in K(\{\{s_0\}\}, (a_1, \ldots, a_{i-1}))$ and $\sigma_i \in k_{a_i}(\sigma_{i-1})$. Concatenating any greedy choice of $\mathbf{b}_i \in R^*(a_i)$ (starting with a_1) such that $f(s_{i-1}, \mathbf{b}_i) = s_i \in \sigma_i$ yields a primitive refinement $\mathbf{b} \in R^*(\mathbf{a})$ such that $f(s_0, \mathbf{b}) \in \sigma_N \subseteq \mathcal{G}$.*

To use sound-intersecting descriptions, we must choose a representation for sets of sets of states. While these sets of sets could potentially grow very large, we can restrict ourselves to more compact but less expressive representations. For example, we can use *factored formulae*, which are tuples (P_s, Φ_s, Φ_i) where Φ_s is a formula over a subset of propositions P_s, and Φ_i is a formula over all propositions. The semantics is that for each combination of values for propositions in P_s consistent with Φ_s, there is some extension to the remaining propositions that is consistent with Φ_i, and represents a reachable state.

Because our example domain (as stated) does not require sound-intersecting descriptions, we only consider ordinary sound descriptions in what follows. However, we expect that in many real domains, sound-intersecting descriptions may be necessary to make interesting (true) assertions about an action's reachable set.

Reasoning with Descriptions

In our implementation, we use DNF formulae to represent sets of states. Thus, to use our sound and complete descriptions, we need to be able to progress and regress sets specified by DNF formulae through NCSTRIPS descriptions. The algorithm for progression is simple: progress

Algorithm 1

```
   function HierarchicalForwardSearch(s_0, G, H)
*      for d = 0, 1, 2, 3, ... do
          for i from 0 to d do
             a ← TopLevelPlan(H, i)
             if FindPrimRef(s_0, G, {a}, d) succeeds then
                return the refinement

   function FindPrimRef(s_0, G, stack, d)
      while ¬IsEmpty(stack) do
         a ← Pop(stack)
         if SucceedsComplete(s_0, a, G) ∧ D(a) ≤ d then
            if SucceedsSound(s_0, a, G) ∧
*              Decompose(s_0, a, G) isn't on the call stack then
               return Decompose(s_0, a, G)
            PushAll(ImmediateRefs(a_i), stack)
      return failure

   function Decompose(s_0, a, G)
      (σ_1, ..., σ_|a|) = Progress(s_0, a)
      (s_1, ..., s_|a|) = Regress(σ, G, a)
      for i from 1 to |a| do
         if IsPrimitive(a_i) then
            b_i ← (a_i)
         else
*           for d = 0, 1, 2, ... do
               r ← ImmediateRefs(a_i)
               b_i ← FindPrimRef(s_{i-1}, {s_i}, r, d)
*              if b_i ≠ failure then break
      return Concatenate(b_1, ..., b_|a|)
```

each clause of a formula separately through each conditional effect of an action, then disjoin the results. To progress a conjunctive clause through a conditional effect, first conjoin the effect conditions onto the clause. If this creates a contradiction, return $false$; otherwise, make all added literals $true$, all deleted literals $false$, and remove propositions from the clause (so that they may take on either value) if known $true$ and possibly-deleted, or known $false$ and possibly-added. Regression can be implemented similarly.

Hierarchical Planning Algorithm

This section presents a top-down, forward search, hierarchical planning algorithm that takes advantage of the semantic guarantees provided by sound and complete descriptions. The algorithm is quite simple, as our focus here is on how to best utilize these descriptions. We expect that speedups will also apply to practical algorithms with heuristics and more sophisticated search control.

Algorithm 1 provides pseudocode for our general hierarchical planning algorithm, which is sound and complete for arbitrary hierarchies. Removing the lines marked (*) and setting d=∞ everywhere yields a simpler version that is correct for non-recursive hierarchies (in which $\forall \mathbf{a} \in \tilde{\mathcal{A}}^*, R^*(\mathbf{a})$ is finite). We begin by describing the operation of this simpler version, and then explain the additions that allow our algorithm to work in recursive hierarchies.

The core of Algorithm 1 is FINDPRIMREF (short for FINDPRIMITIVEREFINEMENT). This function takes in an initial state, a goal set, and a set of high-level plans, and either returns a primitive refinement of some high-level plan that achieves the goal set from the initial state, or fails. HIERARCHICALFORWARDSEARCH simply iterates, in order of increasing length, over top-level plans (with the requirement that each action in the plan be legal in at least one state of the complete set at which it was done), until it finds one for which FINDPRIMREF succeeds.

FINDPRIMREF maintains a stack of plans to consider. At each iteration, SUCCEEDSCOMPLETE progresses the initial state through the complete action descriptions of the next plan on the stack. If the result does not intersect the goal, the plan is not considered further. The vast majority of plans will usually be pruned at this stage. If the plan also SUCCEEDSSOUND (which is checked by progressing the initial state through the sound descriptions and testing if the result intersects the goal), the algorithm commits to this plan and calls DECOMPOSE to decompose the problem of finding a primitive refinement, exiting the outer loop. Otherwise, all immediate refinements of the plan are added to the stack. The function IMMEDIATEREFS can be implemented in various ways, so long as every refinement will be generated exactly once by repeated calls to IMMEDIATEREFS. Our implementation assumes an ordering on the HLAs, picking the first instance of the highest HLA and returning all possible refinements (whose preconditions are satisfied in at least one state in the complete set) in random order. In the example, we used the ordering (MoveBlock, Navigate, Nav).

Finally, DECOMPOSE is called when a plan is found that succeeds soundly. It first computes the sound reachable sets $\sigma_i = H(\{s_0\}, (a_1, \ldots, a_{i-1}))$ at each point in the current plan using PROGRESS. Because the plan succeeds soundly, the last such set $\sigma_{|\mathbf{a}|}$ must intersect the goal. It then uses REGRESS to compute a sequence of states $s_1, \ldots, s_{|\mathbf{a}|}$, where each $s_i \in \sigma_i$. This is done by first choosing any $s_{|\mathbf{a}|} \in \sigma_{|\mathbf{a}|} \cap \mathcal{G}$, then proceeding backwards, choosing each $s_i \in \sigma_i$ such that $s_{i+1} \in h_{a_{i+1}}(s_i)$. For our NCSTRIPS case, this is accomplished using an extension of the progression algorithm, which simply recomputes the progression of σ_i, picking out the first clause that generates a generalization of s_{i+1} and specializing it as necessary to yield a concrete state.[7] We can then solve the subproblems of reaching s_i from s_{i-1} by a refinement of a_i by recursively calling FINDPRIMREF if a_i is high-level (if a_i is already primitive it is guaranteed by the semantics to lead to s_i).

In recursive hierarchies, the algorithm as described thus far could potentially get stuck evaluating infinitely many refinements of a single high-level plan, or repeatedly attempting the same decomposition, and thus fail to return a solution even though one exists. We now describe two additions (marked (*) in Algorithm 1) that prevent these problems, allowing our algorithm to work in general hierarchies.

First, we have added loops over plan *depth limits* around calls to FINDPRIMREF. We define the depth of a plan as the total number of refinements used in generating it within a given call to FINDPRIMREF, and modify FINDPRIMREF to

[7] More generally, we can allow REGRESS to generate formulae describing *all* states (or any subset thereof) that are reachable from the initial state and can reach the goal; computing these formulae might be expensive, but could make the recursive subproblems easier to solve (since they will have multiple goal states).

reject plans $\mathbf{a}$ of depth $\mathrm{D}(\mathbf{a})$ greater than the current limit d. In HIERARCHICALFORWARDSEARCH, we first call FINDPRIMREF with the first possible plan and depth limit 0, then with the first two possible plans with depth limit 1, and so on, ensuring that all possible (finite) refinements of every (finite) top-level plan will eventually be considered. In DECOMPOSE, we discard the original depth limit and again call FINDPRIMREF with iterative deepening, to prevent it from refining the same recursive HLA *ad infinitum*. An alternative would be to have FINDPRIMREF to use a *queue* of plans rather than a stack.

The second addition is a cycle check when calling DECOMPOSE. If the algorithm discovers a soundly succeeding plan inside a call to DECOMPOSE on the same plan in the same circumstances, it does not DECOMPOSE again; instead, it pretends that the plan did not succeed soundly and adds its subplans to the stack accordingly. This ensures that it finds successful refinements of soundly succeeding plans in finite time, preventing the degenerate case in which, e.g., in trying to Nav from (0, 1) to (0, 2), we first decompose into the soundly succeeding plan $(\mathsf{Down}, \mathsf{Nav})$, then decompose that Nav into $(\mathsf{Up}, \mathsf{Nav})$, and so on *ad infinitum* without ever actually making progress towards the goal.

It should be intuitively clear that the algorithm satisfies a completeness property, as the definition of complete descriptions means that a successful plan can never be skipped over. As written, the top level will iterate forever without failing if no plan exists. This can be fixed if an a priori bound is known on the plan length, or if a loopchecker for the planning problem is available.

Theorem 7. *Calls to* FINDPRIMREF *will always terminate. If there exists a primitive refinement of some plan in* $stack$ *that achieves the goal within the depth limit,* FINDPRIMREF *will eventually return a solution. If no primitive refinements of plans in* $stack$ *achieve the goal,* FINDPRIMREF *will eventually fail.*

Remark. If there exist primitive refinements achieving the goal, but none of them are within the depth limit, then FINDPRIMREF may either return one of them or fail.

Corollary 4. *If there exist finite primitive sequences consistent with the hierarchy that achieve the goal of the overall planning problem,* HIERARCHICALFORWARDSEARCH *will eventually return a primitive plan that achieves the goal.*

Since the algorithm uses exact descriptions for primitive actions, it is also sound. But in fact it satisfies a much stronger property:

Theorem 8. *At any stage in Algorithm 1, given that the recursive calls to* FINDPRIMREF *thus far have produced the sequence* $\mathbf{a} = a_1, ..., a_k$*, there are guaranteed to exist successful primitive-level plans that begin by doing the actions in* $\mathbf{a}$*, and* HIERARCHICALFORWARDSEARCH *will eventually return one such sequence.*

This property allows our algorithm to be used by situated agents that interleave planning with execution: as soon as a call to FINDPRIMREF returns, the agent may safely do the corresponding actions without further deliberation.

The algorithm can be extended to use sound-intersecting descriptions, but we have not yet implemented this. The extension requires a more complicated search procedure and definitions of sound and complete success. For the example problem, only sound and complete descriptions are needed.

Inst	Size	Len	F	H	HC	HSC	HSC+
P1	3x4	7	212	80	1	1	0
P2	4x6	48	$>10^4$	$>10^4$	430	135	17
P3	5x8	90	$>10^4$	$>10^4$	$>10^4$	6390	1059

Table 1: Effect of hierarchy and descriptions on running time, rounded to the nearest second. P1, P2, and P3 are instances of the warehouse world. Size refers to the map size and length is the minimum number of steps to achieve the goal. The algorithms compared are F (no hierarchy), H (hierarchy without descriptions), HC (hierarchy with complete descriptions), HSC (hierarchy with sound and complete descriptions), and HSC+ (version of HSC that returns as soon as the first primitive action is found). Algorithm H was allowed to use complete descriptions when determining the set of applicable top-level actions or refinements at each step. Results were averaged across five runs, and each algorithm was stopped if the first run had not terminated within three hours.

Experiments

We have implemented a version of Algorithm 1. Our implementation operates on the aforementioned STRIPS-like representations of planning problems, hierarchies, and descriptions. Table 1 shows running times for several algorithms on some example instances. The absolute numbers should not be taken too seriously, because our DNF operations are not optimized, and a lot of redundant progression is done that could be avoided by caching. Nevertheless, the relative differences between the algorithms are clear.

The flat algorithm ("F" in Table 1) scales poorly as it is just a breadth-first search over primitive plans. The hierarchy without descriptions ("H") does better, but still cannot not handle P2 or P3, as it must consider all refinements (within the current depth limit) of each top-level plan. Adding complete descriptions ("HC") makes a big difference, since most high-level plans are not refined at all. Adding sound descriptions as well ("HSC") provides further gains as the problem size grows, because once a soundly succeeding plan is discovered at the high-level, it can be decomposed into independent navigation subproblems. This decomposability is a general property of navigation, and our algorithm discovers it automatically from the descriptions. The final algorithm ("HSC+") is designed to be used by a situated agent. It returns as soon as it finds a soundly succeeding plan whose first action is primitive. Without sound descriptions, such an algorithm would take as long as the original planner, but here it produces a further speedup.

Related Work

HLA descriptions have been treated in various ways in the literature. One approach is to simply view the descriptions as placing constraints on the planning process: whenever the planner reaches a partial plan that causes any of the HLA descriptions to be violated, it must backtrack. HTNs based on this approach have achieved impressive success in real-world problems (Nau *et al.* 2003). We believe further gains are possible by viewing the descriptions as making specific assertions about the effects of the agent's actions

in the world, and the results of this paper show some of the ways such gains may come about.

Much previous work (Tate 1977; Yang 1990; Russell & Norvig 2003) has assumed that the effect of an HLA is to achieve those literals added by *all* of its refinements. This is, in effect, a limited type of complete description. But as discussed in the introduction, such an adversarial semantics will often prevent us from (correctly) concluding that a high level plan succeeds. Some work (Georgeff & Lansky 1986; Doan & Haddawy 1995) has considered more general forms of complete descriptions, including extensions to metric values, whose representation and use are more similar to ours.

Only one work that we know of (Biundo & Schattenberg 2001) considered sound descriptions, but they do not connect them directly to solution algorithms. In addition, they begin by assuming the descriptions rather than viewing them as logical consequences of the primitive actions and refinement hierarchy.

McIlraith and Fadel (2002) describe a method for synthesizing descriptions of high-level actions that are specified using the Golog language. Their method produces successor state axioms which can, in certain cases, be converted to effect axioms of the sort that we consider. The descriptions will, however, be exact and therefore possibly grow very large for complex actions.

Several researchers (Bacchus & Yang 1991; Knoblock 1991) have studied properties such as ordered monotonicity, the downward refinement property, and the upward solution property. Key differences from our approach are discussed in the introduction.

Discussion and Conclusions

We have demonstrated several classes of HLA descriptions that make specific assertions about the world, and shown theoretically and empirically that they can improve the efficiency of hierarchical planning in complementary ways. There is, however, still much work to be done.

Synthesizing descriptions automatically from the hierarchy is conceptually simple, but raises the difficult issue of automatically managing the complexity-accuracy tradeoff.

There are many possible alternative representations for HLA descriptions and state sets. BDDs (binary decision diagrams) are a promising candidate. In the context of a logic-based hierarchical planner, one might consider using Horn clause approximations. Given appropriate representational choices, incorporating metric values into the framework should be straightforward.

In nondeterministic environments, our complete descriptions may work unchanged, whereas sound-intersecting descriptions will be the natural analogue of sound descriptions. In fact, intersecting descriptions and many instances of nondeterminism have similar motivations: we replace a complicated deterministic dependency with uncertainty.

For practical applications, it will be essential to extend current domain-independent heuristics to handle HLAs with sound and complete descriptions. Especially when descriptions are represented in a simple language (e.g., NCSTRIPS), it should be easy to incorporate them into existing heuristics (e.g., planning graphs). When a high-level plan does not fail completely or succeed soundly, HLA descriptions may also help us decide which action to refine next.

Finally, many existing hierarchical planning systems use refinements into partially ordered rather than totally ordered subplans. One might, therefore, worry that nothing interesting and true can be said about an HLA in this context, since any other action might be interleaved with its expansion, disrupting its effects (Young, Pollack, & Moore 1994). This concern is based, implicitly, on an adversarial semantics for HLAs. In fact, our sound descriptions should work unmodified in the partially ordered case, since the planning agent can always choose not to interleave the refinements of different HLAs. In contrast, extending complete descriptions to work in this setting may be more challenging.

Acknowledgements

Bhaskara Marthi thanks Leslie Kaelbling and Tomas Lozano-Perez for their support during the latter part of this research. This research was also supported by DARPA IPTO, contracts FA8750-05-2-0249 and 03-000219.

References

Bacchus, F., and Yang, Q. 1991. The downward refinement property. In *Proc. IJCAI '91*, 262–292.

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief – A preliminary report on combining state abstraction and HTN planning. In *Proc. ECP-01*, 157–168.

Doan, A., and Haddawy, P. 1995. Decision-theoretic refinement planning: Principles and application. Technical Report TR-95-01-01, Univ. of Wisconsin-Milwaukee.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3/4):189–208.

Georgeff, M. P., and Lansky, A. L. 1986. Procedural knowledge. *Proc. IEEE* 74(10):1383–1398.

Jagadeesan, R.; Shanbhogue, V.; and Saraswat, V. 1992. Angelic non-determinism in concurrent constraint programming. Technical report, Xerox PARC.

Knoblock, C. A. 1991. *Automatically generating abstractions for problem solving*. Ph.D. Dissertation, Carnegie Mellon University.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

McIlraith, S. A., and Fadel, R. 2002. Planning with complex actions. In *Proc. NMR '02*, 356–364.

Nau, D.; Au, T. C.; Ilghami, O.; Kuter, U.; Murdock, W. J.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition.

Simon, H. 1962. The architecture of complexity. *Proc. American Philosophical Society* 106(6):467–482.

Tate, A. 1977. Generating project networks. In *Proc. IJCAI '77*.

Yang, Q. 1990. Formalizing planning knowledge for hierarchical planning. *Comput. Intell.* 6(1):12–24.

Young, R. M.; Pollack, M. E.; and Moore, J. D. 1994. Decomposition and causality in partial-order planning. In *AIPS '94*.