HOPPER: a hierarchical planning agent for unpredictable domains

Maciej Wojnar **Peter Andreae**

School of Mathematics, Statistics and Computer Science
Victoria University of Wellington,
PO Box 600, Wellington 6140, NEW ZEALAND,
Email: Maciej.Wojnar@mcs.vuw.ac.nz
Email: Peter.Andreae@mcs.vuw.ac.nz

Abstract

Hierarchical Task Networks (HTNs) are a family of powerful planning algorithms that have been successfully applied to many complex, real-world domains. However, they are limited to predictable domains.

In this paper we present HOPPER (**H**ierarchical **O**rdered **P**artial-**P**lan **E**xecutor and **R**e-planner), a hierarchical planning agent that produces partial plans in a similar way to HTNs but can also handle unexpected events in unpredictable domains by interleaving planning and execution. HOPPER can detect and recover from unexpected events that invalidate the plan, and it can detect and exploit unexpected opportunities both serendipitously and by interleaving decompositions.

Keywords: Planning, HTN, Reactive Agents, Goal Decomposition

1 Introduction

Creating an intelligent agent that can plan with foresight and behave intelligently has been one of the central problems in Artificial Intelligence since the field's beginning. In particular, the problem of creating a planning agent that performs well in domains and problems which humans excel at (which we will refer to as the Human Planning Domain, or HPD) has been the focus of much research. The classical approach has been to develop algorithms to solve the planning problem in simplified "toy" domains, and then gradually try to scale these solutions back to the complexity of the HPD.

Hierarchical Task Networks (covered in section 2) are a family of planning algorithms that handle the complexity found in the HPD by encoding domain knowledge in hierarchical rules for achieving goals or tasks. HTNs have been quite successful in scaling to a variety of complex domains, but because they keep the planning and execution phases separate, they face the same problems that classical planners do when scaling to partially-observable, non-deterministic, and not fully-controllable domains more similar to the HPD.

This research was supported in part by a 2005 Bright Future Top Achiever Doctoral Scholarship.

In this paper, we present HOPPER, a planner that, like HTNs, stores domain knowledge in hierarchical rules and like HTNs, scales well to more complex domains. However, HOPPER interleaves planning and execution, allowing it to robustly respond to unexpected events during the execution of its plans. Specifically, HOPPER detects unexpected failures (unexpected events that invalidate its current plan) early, and adjusts its plan during execution. HOPPER also detects and exploits unexpected opportunities (unexpected events that allow the goal to be accomplished more efficiently) by adjusting its plan during execution.

1.1 Outline of the Paper

The rest of this paper is organized as follows:

- Section 2 briefly describes the problem of planning in complex domains and gives an overview of HTN planning, explaining how it handles complexity by encoding domain knowledge in hierarchical rules.
- Section 3 describes the problem of planning in unpredictable domains.
- Section 4 describes HOPPER in detail and explains how it handles unpredictable domains and the twin challenges of unexpected failures and unexpected opportunities.
- Section 5 concludes the paper by discussing the kinds of domains HOPPER is most suited to and discusses future extensions to HOPPER.

2 Hierarchical Task Networks

Classically the planning problem has been construed as the problem of generating a sequence of atomic actions that, when executed in order, transform an initial state to a final state that satisfies some goal constraint(s).

An example of a classical planner is STRIPS (Fikes & Nilsson 1971) where a state consists of a collection of ground atoms; for example:

isMovable(cup1), isSurface(table1), isSurface(floor1), on(cup1, table1)

An action specifies which atoms are added to a state and which are deleted given that some precondition holds; for example:

```
move(X,Y,Z)
  PRE: isMovable(X), on(X,Y), isSurface(Z)
  DEL: on(X,Y)
  ADD: on(X,Z)
```

A goal state specifies the atoms that must be true and the atoms that must not be true; for example:

```
MUST: on(cup1,floor1)
MUST NOT: dirty(cup1)
```

STRIPS searches for a sequence of actions that will transform an initial state into a goal state, using only the knowledge expressed in the action descriptions. The problem with planning from an initial state to a goal state(s) is that in general, a state can be transformed into multiple other states, and a state can also be reached from multiple other states. This makes domain-independent planning completely intractable for all but the simplest problems (Erol et al. 1995).

It is necessary to use domain knowledge to guide the search in order to be able to scale to domains with a greater number of possible actions and more complex states. A particularly powerful method is to encode the domain knowledge in hierarchical decomposition rules.

2.1 Decomposition Rules

Planning with with hierarchical task networks (HTNs) has been around for some time (for example (Wilkins 1984)), but it was not until 1994 that HTN planning was formalized (Erol et al. 1994) and an HTN algorithm (UMCP) was presented that was provably sound and complete.

Hierarchical Task Networks can use a representation for states, goals and actions similar to that of STRIPS [1]. The fundamental way that HTNs differ from classical planning algorithms is in how they use domain knowledge to guide the search.

The domain knowledge of an HTN planner consists of a number of decomposition rules, where each rule specifies how a goal can be achieved by achieving a sequence of sub-goals. Usually a decomposition also has a precondition specifying in what states it is applicable. For example, a rule for delivering a package in a logistics domain could look like:

```
achieve atLocation(P, L1)
  PRE: isPackage(P), atLocation(P, L0),
  isLocation(L1), isTruck(T) →
    achieve atLocation(T,L0),
    achieve in(P,T),
    achieve atLocation(T,L1),
    achieve ¬in(P,T)
```

A single decomposition rule can encode indefinitely long plans if the goal it achieves also occurs in its sub-goals. For example, the following pair of decomposition rules describe how to get hold of an object:

```
achieve have(X)
  PRE: in(X,Y) →
    achieve open(Y), achieve have(X)
  PRE: ¬in(X,Y) →
    achieve nextTo(hand, X),
    achieve isGrasping(hand, X)
```

This pair of rules is analogous to inductive rules with one specifying the inductive case of what to do when the desired object is in a container, and the other specifying the base case of what to do otherwise. These two rules will produce the appropriate plan for getting an object out of indefinitely many nested containers.

[1] Though some HTN planners (UMCP among them) have enriched the definition of a goal to distinguish such goals as *having* a house and *building* a house, this distinction is not important in this paper.

2.2 Planning with Decomposition Rules

The HTN planning algorithm searches backward from the final goal, but rather than searching through atomic actions that achieve states, it searches through decompositions that achieve the main goal, then tries to find decompositions for achieving the sequence of sub-goals, and the sub-sub-goals and so on until reaching atomic goals achievable with a single action. The algorithm continues until there are no more non-atomic goals to achieve and the sequence of atomic goals can be converted directly into a sequence of atomic actions. Depending on the specific kind of HTN, the algorithm may also process the final sequence of atomic actions to resolve any conflicts, bind variables, etc.

2.3 Ordered HTNs

(Nau et al. 1999) found that constraining the HTN search to decompose sub-goals in the same order that they will later be executed allows the use of more expressive preconditions — if goals are decomposed in the order they are executed, then all of their variables are already bound, and the intermediate state corresponding to that stage of the planning process can be determined.

If the intermediate states are completely known, then sophisticated preconditions and reasoning can be applied to those states. For example, a sub-goal to be at some destination (*e.g.* to be at the airport as part of the plan of going on a trip) can be achieved with decompositions for walking there, calling a taxi, catching a bus and so on. Which of these decompositions is appropriate or even possible depends crucially on where the agent will be at that stage of the plan:

```
achieve atLocation(X)
  PRE: at(Y), walkingDistance(Y,X) →
    Action: Walk(X)
  PRE: at(Y), enoughMoneyForTaxi(Y,X) →
    at(taxiStand), hailedTaxi(T), in(T),
    giveDirections, exit(T)
```

Though completely ordering the decompositions allowed for powerful, expressive preconditions, (Nau et al. 2001) found that the algorithm was limited by not being able to interleave sub-goals properly. For example, when achieving the goal of having two packages at the same location in a logistics domain, the planner would return a sub-optimal plan of picking up each package and dropping it off separately rather than combining both activities into one shorter, more efficient plan.

To get around this problem, they extended the algorithm so that the sub-goals of a decomposition could be arranged in a partial-order, and the decompositions of co-ordered sub-goals could be interleaved to produce a more efficient plan. While this approach does allow the planner to interleave the immediate sub-goals of two co-ordered goals, it depends on each decomposition being hand-coded to specify how its sub-goals can be interleaved.

2.4 Real-world applications of HTNs

Because decomposition rules offer a powerful and intuitive framework for encoding domain knowledge, Hierarchical Task Networks have seen extensive real-world use. This includes planning the electronic

and mechanical design of microwave modules (Hebbar et al. 1996), planning the declarer play of contract bridge (Smith et al. 1996), planning noncombatant evacuation operations (where the goal decomposition is human-supervised) (Muoz-Avila et al. 1999), and the automated composition of web services (Wu et al. 2003).

2.5 HTNs in non-deterministic domains

Though most HTN applications have been in deterministic, fully-observable, and fully-controlled domains, there have been a number of extensions where some of these constraints have been relaxed. (Kuter et al. 2005) and (Kuter & Nau 2006) extended the classical planning model so that the result of atomic actions could not be deterministically predicted (whether due to random environmental influences or the actions of other agents). However, while the result of an atomic action was extended from a single possibility to a set, the set of possibilities was constrained to be a finite and *known* set.

For example, the action of picking up a block may non-deterministically result in the block being picked up or it being on the ground (because it slipped out). Or the action of moving toward a prey agent may non-deterministically result in the prey agent moving to any one of its 8 adjacent squares. The crucial constraint is that all the possible results of each action are known and no other results are possible.

HTNs have also been extended to partially observable domains (Kuter et al. 2007), but again the unknown variables in those states and their possible values are known.

Contingency planning is most appropriate in such domains where all of the possible results of an action can be predicted. However, these methods do not scale to a truly unpredictable domain where the set of possible states following the execution of an action is unknown.

3 Planning in an Unpredictable Domain

The standard approach to the planning problem is to find a sequence of atomic actions that will achieve the goal and then blindly execute them.

This approach is appropriate to a fully deterministic, completely observable and completely controllable domain (for example, Blocks World) but it runs into problems in a domain where the results of actions are not deterministic, the state of the world is not observable (*e.g.* the agent cannot tell whether a container is locked until the agent attempts to open it), or where the state of the world is not completely controllable by the agent (*e.g.* the locations of various objects in the world may change due to the actions of other agents): in general, when the future state of the world is unpredictable.

The HPD has all three of these characteristics and any agent that acts intelligently in such a domain must be able to deal with unpredictable events.

3.1 Classical Plans are Brittle

When producing a sequence of atomic actions, a planner implicitly makes predictions about what the state will be when each of the actions will be executed. If an unexpected event causes any of these predictions to fail, then the entire plan is invalidated and the agent must re-plan from the new state. The longer the plan the higher the chance that one of its predictions will fail making it more brittle and unreliable.

Contingency planning helps to ameliorate this problem by trying to predict in advance where the plan may fail and generating backup plans that will still achieve the goal. This approach is fundamentally limited by the need of knowing all the ways that a plan could fail, but in the HPD this is an impossible constraint to satisfy.

There have been previous approaches that interleave classical planning and execution. (Haigh & Veloso 1998) developed a robotic system that uses means-end analysis to generate plans. To try to deal with the uncertainty of the world, it executes actions as soon as all of their preconditions are satisfied even when the plan itself is not complete. Though this worked reasonably well with high-level, navigational plans (e.g. go to office1, pick-up mail, go to office2, deliver mail), it is not clear that this approach can scale to more complex, longer plans in richer domains. This approach also has difficulty dealing with unexpected failures and opportunities, only being able to notice failures of individual actions, and opportunities of serendipitously achieved preconditions. Again, it is not clear that this approach can scale to handle more complex and abstract failures and opportunities.

In general, it is dangerous to start executing a classical plan before it is complete because there is no guarantee that the partial plan is even feasible. For example, in goal-oriented planning, until the plan is complete, there is no way to know whether it is safe to execute an action in the plan, since the incomplete part of the plan may require some other action to be executed first.

3.2 Unexpected Failures

During execution, the unexpected invalidity of a plan is normally detected at the moment of plan failure: when the precondition of one of the atomic actions fails. However the event invalidating the plan can occur much earlier in the execution of the plan, and the sooner it is detected, the better. This is because the actions after the invalidating event are no longer guaranteed to lead the agent closer to the goal. These actions will, at best, be an inefficient waste of time, and at worst, can lead to a state that the agent cannot recover from. For example, if during the execution of a plan to fly to another country the agent discovers that its flight will leave an hour earlier, but the agent does not identify the invalidity of its plan until it tries to board the non-existent plane at the airport, then there will be no easy way for it to re-plan to achieve the original goal because by that time the plane will have already left.

In a dynamic domain where unexpected, usually innocuous, events are constantly occurring, it is difficult to determine whether a given event will invalidate the plan, and so the agent is forced to constantly simulate executing the rest of its plan to make sure that it is still viable. Furthermore, because each atomic action in the sequence depends on the state in which it will be executed which in turn depends on all of the previous actions, altering an action will usually invalidate the rest of the plan. In general, when a plan is invalidated, then the agent has to re-plan from scratch as there is no real way to re-use the remaining invalidated action sequence.

3.3 Unexpected Opportunities

A more insidious problem that classical planners face in an unpredictable domain is that of unexpected opportunities. These events change the state in such a way as to make it possible for the goal to be achieved more efficiently (*e.g.* with a fewer number of atomic actions). However, because these actions do not invalidate the original plan, they are very hard for a

classical planner to detect, let alone exploit. For example, if during the execution of a plan to fly to another country the agent discovers that a friend is driving to the airport at the same time, then the agent should somehow detect and exploit this opportunity even though its original plan of calling a taxi has not been invalidated.

3.4 Reactive Agents

An approach to handling non-deterministic, unpredictable domains has been to make the agent completely reactive. Reactive agents (for example, (Agre & Chapman 1987)) constantly monitor the state and execute only a single action at a time. The action to be executed is determined based on the current state and possibly the top level goal the agent wants to achieve. However, since all possible facts in the world may be relevant to the choice of action now, for all but the simplest goals, it is not feasible to construct a comprehensive policy or set of reactive rules. For example, the fact that the agent's car had an accident yesterday is highly relevant to the decision of opening a suitcase if the agent's goal is to go on a road trip, but it is irrelevant to the same decision if the agent's goal is to unpack from a trip it just came back from.

4 HOPPER

In an unpredictable domain, there is always a degree of uncertainty about the current state and any future predicted state, and the more distant a predicted state is in the future the more uncertain it is. This is because there is a longer stretch of time in which an unpredictable event could occur that would invalidate the prediction. Because HTNs separate planning and execution, a plan they produce has to anticipate all of the possible events that could disrupt it during execution. This is an impossible task in a truly unpredictable domain like the HPD.

HOPPER addresses this property of unpredictable domains by producing a plan with varying degrees of abstraction. The earlier, more predictable parts of its plans start with ground, atomic, directly executable actions. The later parts of its plans become increasingly more general, reflecting the increasing uncertainty of future states. This allows HOPPER to interleave planning and execution letting it flexibly respond to unexpected events.

HOPPER is a hybrid of planning and reactive agents. It shares many similarities with HTN planning agents and itself produces partial plans. However it also constantly monitors its environment and adapts its behaviour to any changes, in a similar way to reactive agents.

4.1 Algorithm

At regular, fixed intervals HOPPER receives information about the current state of the world. Every time that it receives this information, HOPPER updates its partial plan, determines the appropriate atomic action to execute and executes the action. The core of the HOPPER algorithm is the **GetAction** function which is called every time that HOPPER needs to perform an action. The function recursively decomposes high-level abstract goals into simpler subgoals until it reaches atomic goals which it can achieve with a single action. Figure 1 provides a high-level, abstract description:

The rest of this section will explore this algorithm in detail.

```
GetAction(Goal)
  if goal satisfied in current state
    if goal = top goal return SUCCESS
    otherwise return GetAction(parent of goal)
  otherwise
    if goal is atomic return GetAtomicAction(goal)
    otherwise if goal has sub-goals
      return GetAction(first sub-goal(s))
    otherwise if goal is decomposable
      select decomposition and decompose goal
      return GetAction(goal)
    otherwise if goal = top goal return FAILURE
    otherwise
      remove decomposition from parent of goal
      return GetAction(parent of goal)
```

Figure 1: High-level description of GetAction function

4.2 Representation

HOPPER represents goals as properties of objects and relationships between objects that must and must not hold and represents decompositions as partially-ordered sequences of sub-goals in a similar way to HTNs.[2]

4.3 Decomposition Strategy

HOPPER decomposes a goal by selecting an applicable decomposition rule and producing the corresponding partial-order of sub-goals. It continues by decomposing the sub-goals in a similar way, but only those sub-goals that are not constrained to come after any another sub-goal. If a decomposition is completely ordered, then only its first sub-goal will be decomposed; if it is completely unordered, then all of its sub-goals will be decomposed. HOPPER continues to recursively decompose the unconstrained subgoals according to this "left-fringe-first" decomposition strategy until it reaches atomic goals that can be achieved with a single action.

The hierarchy of goals produced by the "left-fringe-first" decomposition strategy can be viewed as a partial-plan of increasing generality: the order of the leaf nodes of this hierarchy will produce a sequence of sub-goals to be achieved that is increasingly general. In this case, by generality we mean not the set of states in which the goals are satisfied but the set of states in which they are *achievable*.

4.3.1 Decomposition by Least Commitment

In general, different decompositions of the same goal will correspond to different ways of achieving the goal. The different decompositions will also be applicable in different sets of states. Selecting a decomposition with which to decompose a sub-goal is equivalent to making a commitment about what the state will be when the goal is achieved. Subsequent decompositions lower in the hierarchy further constrain and restrict the set of states in which the top sub-goal will

[2]HOPPER is partially integrated with a decomposition rule learner, and the goals and rules it uses are learned and so contain counts and probabilities on their atoms, variables, ordering, etc. Because of this, when achieving a new goal, HOPPER must first match it against its known goals to find the appropriate decompositions to use; and when HOPPER checks whether or not a goal is achievable or satisfied, it must match the goal to the current state. However, because the rule learner is not yet complete and HOPPER works just as well with the standard, less flexible HTN representation, we have omitted a detailed discussion of the representation.

be achieved. By not decomposing future sub-goals, HOPPER does not commit to what the state will actually be when those sub-goals are achieved. The "left-fringe-first" decomposition strategy is analogous to a least commitment search.

It is important to note that the "left-fringe-first" decomposition strategy is completely compatible with ordered HTNs (see section 2.3). It makes fewer commitments than a fully ordered decomposition strategy, but it can still make full use of the sophisticated preconditions made possible by ordered HTNs.

Figure 2 shows an example of a left-fringe decomposed goal hierarchy for having lunch. The rectangles represent goals and sub-goals that HOPPER tries to achieve. Co-ordered sub-goals, sub-goals that may be achieved in arbitrary order, are surrounded by a dashed oval. The lowest level sub-goals can be achieved by executing a single action, represented by a filled oval. The white rectangles are goals that have been decomposed already. The shaded parts of the decomposition hierarchy represent the plan of increasing generality that HOPPER will try to achieve. The dashed arrows show the order that HOPPER will try to achieve the leaf sub-goals (whether by decomposing them into sub-sub-goals or into atomic actions).

Note that because the goals "achieve tea" and "achieve sandwich" are co-ordered, HOPPER would actually decompose the left-fringe of both sub-goals. However, for the sake of clarity, we have omitted the decomposition of the "achieve sandwich" sub-goal.

4.3.2 Robustness against Unpredictable Events

The more general a sub-goal to be achieved, the more robust it is against unpredicted events (i.e. the less likely it is that an unpredicted event will alter the state in such a way as to make the sub-goal unachievable). Plans consisting of increasingly more general goals to achieve are perfectly suited to an unpredictable domain with its increasingly more uncertain future states. Because of this match-up of general goals to uncertain states, many of the unpredicted events that would cause a classic plan to fail are dealt with automatically by the simple fact that the affected sub-goals have not yet been decomposed. By not decomposing future sub-goals until it is necessary, HOPPER can also opportunistically take advantage of unexpected events that allow a better decomposition to be used to achieve the sub-goal.

In an unpredictable domain, the advantage of a least commitment strategy of decomposing sub-goals is that for uncertain, future sub-goals it does not overcommit to a specific decomposition hierarchy which may end up being invalidated or may prove to be suboptimal.

For example, when planning an international trip, the unexpected event of discovering you do not have enough money to pay for a taxi to get you to the airport would not affect a robust plan where the goal of getting to the airport had not yet been decomposed (in this case it could later be decomposed to catching a bus). Similarly, the unexpected event of discovering that a friend can drop you off at the airport would also not change the decomposition hierarchy. When it became necessary to decompose the goal of getting to the airport, the superior decomposition of having a friend drop you off would be used instead.

4.4 Backtracking to Parent Decompositions

As HOPPER decomposes the left-fringe of the goal decomposition hierarchy, if it reaches a goal that is unachievable (whether because none of its decomposition rules are applicable in the current state, or because they have already been tried) then it backtracks to the parent decomposition that spawned it and re-decomposes it with another applicable decomposition, and if no such decomposition exists then the parent decomposition fails and HOPPER continues backtracking.

If HOPPER reaches a goal that is not achieved in the current state but all of its sub-goals are, then it re-decomposes that goal with the *same* decomposition. This means that when executing a decomposition repeatedly fails to achieve its goal, HOPPER will continue to execute it. This has the advantage that actions (whether atomic or higher-level) that need to be repeated an indefinite number of times to achieve a goal can be encoded within a single decomposition rule. For example, you keep stirring a cup of tea until the sugar is dissolved.

To get around the problem of HOPPER futilely executing a possibly incorrect decomposition indefinitely, HOPPER keeps a count of the number of times it has attempted to execute the decomposition in a row. If this count goes above a certain threshold, then HOPPER recognizes the decomposition as having failed and backtracks normally. This is a rather crude mechanism, but it can be extended so that each decomposition has its own threshold of how many times it should be attempted before being abandoned (*e.g.* once or twice for opening a container, for a minute when stirring tea, or many times over many months when nagging a PhD student to write a paper).

4.5 Detecting and Handling Unexpected Events

After decomposing the left-fringe of the goal decomposition hierarchy, HOPPER selects the leftmost atomic goal to achieve and determines the grounded atomic action that will achieve it. Just prior to executing the action, HOPPER makes a prediction of how the state will change as a result of the action. The prediction is generated by adding the "must" properties and relations specified in the atomic goal to the current state and deleting the "must not" properties and relations as well as applying any other conditional effects specified by the decomposition. HOPPER then executes the atomic action and compares the resulting state to its prediction.

HOPPER treats any discrepancy between the predicted state and the actual state as being due to an unexpected event. It determines which objects unexpectedly changed their properties or relationships and then traverses the goal decomposition hierarchy to ensure that the sub-goals dependent on these objects are still valid.

4.5.1 Plan Failure

In general, HOPPER will detect an event that invalidates its plan as soon as it occurs. The most common way that a sub-goal becomes unachievable is by some change to an object that is involved in the sub-goal (for example, if the agent discovers that its car has been stolen, then a future sub-goal involving the car may now invalid). To detect this kind of failure, HOPPER checks each sub-goal in the hierarchy to make sure that it is still applicable in spite of the unexpected change to the object. If a sub-goal is no longer applicable, then the parent goal has to be re-decomposed in another way.

Limiting plan failure to the re-decomposition of the parent goal of the offending sub-goal allows HOPPER to re-use almost all of the remaining plan and removes the need for re-planning from scratch.

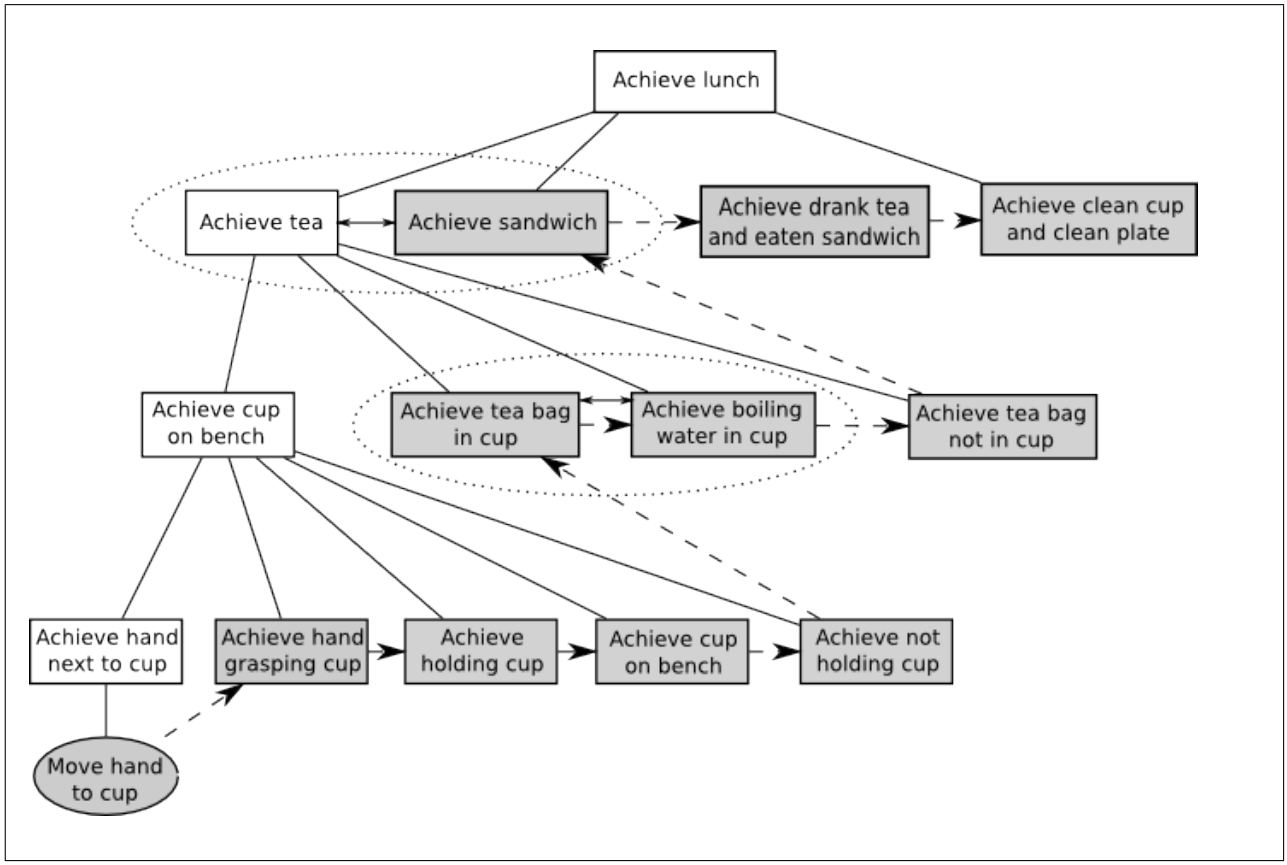

Figure 2: Decomposition Hierarchy for having lunch

4.5.2 Opportunities

There are two kinds of opportunities that HOPPER can exploit: serendipitous opportunities for achieving future abstract sub-goals that have not yet been decomposed (see section 4.3.2), and interleaving shared sub-goals of consecutive co-ordered goals.

If an event occurs that makes it easier to achieve a future as yet undecomposed sub-goal HOPPER will automatically take advantage of it when it comes time to decompose it. Sometimes a future sub-goal may become serendipitously achieved, and because it does not matter whether or not the sub-goal was achieved by a decomposition or by some external agency (except in the case of "clean-up" sub-goals: see section 4.6) the difference is transparent to the algorithm. For example, if HOPPER is trying to achieve the goal of having a cup of tea, then a sub-goal that it needs to achieve is to have boiling water. If, when the agent goes into the kitchen to put the kettle on to boil, it notices that there is already a kettle of boiling water on the stove, then it will use this kettle rather than continuing with the decomposition of putting another kettle of water to boil independently.

The second way that HOPPER optimizes its behaviour is by interleaving sub-goals that are co-ordered (neither is constrained to be before the other) in a decomposition.

When the left-fringe of the goal decomposition hierarchy is decomposed (or re-decomposed in the case of an unexpected event), co-ordered sub-goals are decomposed in parallel. HOPPER then traverses the sub-hierarchies of the co-ordered goals and identifies any shared sub-goals that they may have. After identifying two shared sub-goals, HOPPER attempts to interleave the two decompositions the shared sub-goals belong to. If it can find a consistent interleaving of the two decompositions then it remembers this sequence and executes the sub-goals according to this order, overriding the normal depth-first order of the decomposition hierarchy until all of the interleaving sub-goals have been achieved or an unexpected event disrupts the hierarchy.

Figure 3 shows an example of interleaving the decompositions for delivering two packages in a logistics domain, one locally by truck, and one to a more distant location by plane. Because these two goals can be achieved in either order, HOPPER will decomposes the left-fringe of both.

The two decompositions share a sub-goal ("Achieve truck1 at loc1", the highlighted rectangles in both decompositions) and so HOPPER will interleave the two decompositions where the shared sub-goal was found. When interleaving two decompositions, HOPPER attempts to order the sub-goals in such a way that the sub-goals of each decomposition remain in the same order that they were decomposed, the shared sub-goals are executed together, and the sub-goals of one decomposition do not interfere with the sub-goals of the other. A sub-goal interferes with another decomposition if achieving it would falsify an earlier sub-goal from the other decomposition (unless it is already falsified or about to be falsified by a sub-goal from the same decomposition).

In the example above, HOPPER would first fix "Achieve truck1 at loc1" and then interleave the remaining three sub-goals of both decompositions. The sub-goal "Achieve truck1 at loc2" falsifies the earlier sub-goal "Achieve truck1 at loc1" in the other decomposition. Because of this, "Achieve truck1 at loc2" cannot be achieved until after the sub-goal "Achieve truck1 at airport" becomes available in the other decomposition (as this would falsify "Achieve truck1 at loc1"). This results in an interleaving where both packages are loaded into the truck first before being delivered to their respective destinations.

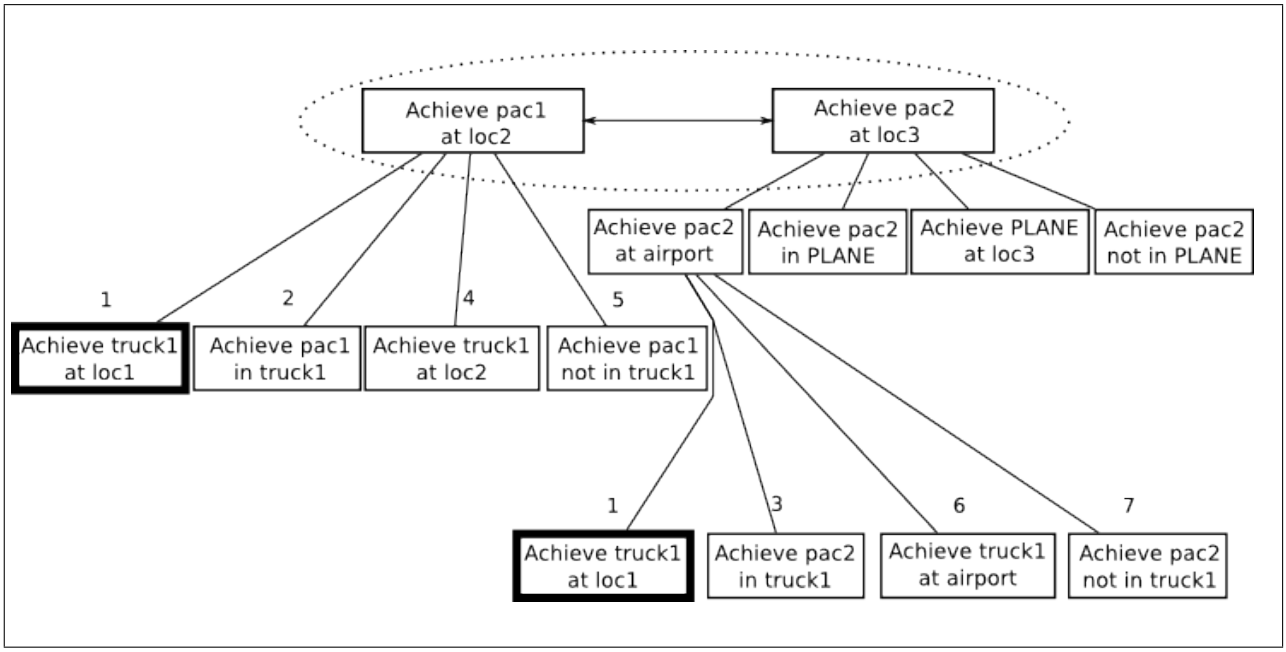

Figure 3: Interleaving the delivery of two packages

In this way, HOPPER can interleave decompositions automatically without needing the decompositions to be hand-coded with any extra information. If multiple co-ordered goals share the same sub-goal, then HOPPER can interleave multiple decompositions in a similar way.

HOPPER can also take advantage of interleaving opportunities if an unexpected event causes them to suddenly become possible. For example, if HOPPER mistakenly believed that pac2 was at a different location (*e.g.* loc4), then the two decompositions would not share any sub-goals and would not be interleaved. But if upon arriving at loc1 (while achieving "pac1 at loc2") HOPPER discovered that pac2 was in fact at loc1, then this would invalidate the "achieve pac2 at loc3" sub-goal (which also records pac2's initial location) and HOPPER would re-decompose it, discover the now shared sub-goal and interleave the two decompositions as above.

4.6 "Clean-up" Sub-Goals

HOPPER assumes that its decompositions will minimize their side-effects, making them predictable and modular (decompositions for achieving a sub-goal can be interchanged depending on the state without needing to alter the rest of the decomposition hierarchy).

The decompositions that HOPPER uses minimize their side effects by labeling some of their sub-goals as "clean-up". In order to be satisfied, a goal in the hierarchy must not only match the current state but it must also not have any "clean-up" sub-goals in its decomposition. If it does, then the algorithm continues to achieve the goal's sub-goals until all of its "clean-up" sub-goals are satisfied. The "clean-up" sub-goals usually occur after the decomposition's goal has been achieved, and their sole purpose is to minimize the side-effects of the decomposition. For example:

achieve on(X, Y)
  PRE: clear(X), on(X,Z) →
  achieve at(hand, X),
  achieve isGrasping(hand, X),
  achieve ¬on(X, Z),
  achieve on(X, Y),
  achieve ¬isGrasping(hand, X)

As soon as the hand places cup X on saucer Y the top goal is achieved. However, the hand is still holding cup X. This is a side-effect of the decomposition and needs to be undone by achieving ¬isGrasping(hand, X) before going on to other decompositions.

achieve atLocation(P, L1)
  PRE: isPackage(P), atLocation(P, L0),
    isLocation(L1), isTruck(T) →
  achieve atLocation(T,L0),
  achieve in(P,T),
  achieve atLocation(T,L1),
  achieve ¬in(P,T)

As soon as the truck arrives at the destination the top goal of the package being at that destination will be achieved. However, the package is still inside the truck. This is a side-effect of the decomposition and needs to be undone by achieving ¬in(P,T).

Minimizing the side-effects of decompositions in this way makes it less likely that achieving future goals will undo goals already achieved. It makes the decompositions more predictable and more modular. It also tends to makes future sub-goals easier to achieve.

5 Conclusion and Future Work

HOPPER has all the advantages of HTNs over classical planning, but is able to deal with unpredictable domains where the set of possible things that could happen cannot be enumerated. By producing a plan that increases in abstractness with distance from the present, it avoids prematurely committing to decisions about future states.

There are two aspects to an evaluation of HOPPER: determining the range of problems on which it is effective, and measuring its performance characteristics. We have applied HOPPER to a variety of smaller planning problems in a transport domain and a kitchen domain which have demonstrated that HOPPER's mechanisms for dealing with unpredictable events and opportunities work effectively on

small problems. We have yet to apply it to large problems and other domains, which would be required for an effective performance evaluation.

Because HOPPER is a kind of HTN planner, it could be used on the same range of domains as other HTN planners — any domain for which effective goal decomposition rules can be constructed. It differs from other HTN planners in that it cannot guarantee soundness or completeness of the plans, because it necessarily makes commitments to actions before the plan is complete.

In deterministic domains, because HOPPER can interleave co-ordered sub-goals, it is able to produce more efficient plans than standard HTN methods. In non-deterministic domains, its interleaving can also take advantage of unpredictable opportunities, which no other HTN methods do. We are not yet able to characterise the limits on the range of problems for which HOPPER is able to perform effective interleaving.

The main advantage of HOPPER lies in being able to handle unpredictable domains where committing to details of distant parts of the plan is unjustified due to the uncertainty of the world, but where the goals are sufficiently complex that it is necessary to look ahead in order to choose actions sensibly. Because HOPPER does not commit to future parts of the plan until it has done enough actions to determine the state of the world, it is able to detect and recover from disruptive unexpected events which would destroy plans of ordinary HTN planners. It is also able to detect and exploit unexpected opportunities both serendipitously and by interleaving decompositions at any level of abstraction. Because ordinary HTN planners must complete the plan before any execution, they are unable to do this.

In principle, classical planners (or HTNs) can deal with unpredictable domains by re-planning completely after every action. However, this is extremely inefficient. HOPPER is much more efficient because it does not need to reconstruct the entire plan when unpredictable events occur.

5.1 Future Work

HOPPER is part of a larger project for learning decomposition rules for routine behaviour in complex, unpredictable domains. We are currently working on the rule learning mechanism; however, we believe that HOPPER alone provides a good framework upon which more sophisticated reasoning and planning mechanisms can be built:

- We are in the process of adding decomposition knowledge specifying how many times a given decomposition can be repeated before failing.
- The fact that sub-goals are not constrained to be in a particular decomposition does not mean that the order they are achieved in does not matter. For example, it is better to deliver a package locally first and internationally second so that the agent does not end up overseas when it wants to deliver the first package. We are extending HOPPER so that it can reason about the optimal ordering of unconstrained, unordered sub-goals.
- If a sub-goal fails because none of its decompositions are applicable, then, rather than failing and backtracking, HOPPER could be extended to posit a new sub-goal of achieving one of the required preconditions.
- The decomposition strategy could be tailored to the requirements of the domain. Rather than only decomposing when necessary, HOPPER could be extended to look ahead more; particularly important sub-goals could be decomposed earlier and deeper to make sure that they will be achievable.
- Decompositions have preconditions and effects and are in effect high-level actions. This means that they can be put into a classical planner to produce a high-level plan. If no decomposition can be found to achieve a sub-goal, then HOPPER could be extended to plan out a decomposition using sub-decompositions from similar goals.
- HOPPER can be extended to have probabilistic knowledge about what is true in the current state. It could be extended to have explicit sensing actions and sensing sub-goals for making sure that critical aspects of the state hold.
- Currently all events not caused by the agent are unexpected. HOPPER could be equipped with knowledge about how the environment works to be able to better predict the future, and to be able to exploit the environment to achieve some sub-goals (*e.g.* waiting until something is cooked).
- HOPPER currently has only a single hierarchy where all the sub-goals an agent achieves are a means to a single top goal. However, an agent may have multiple, independent, even conflicting goals at once. HOPPER could be extended to prioritize among multiple goal decomposition hierarchies.

References

Agre, P. E. & Chapman, D. (1987), Pengi: An implementation of a theory of activity, *in* 'Sixth National Conference on Artificial Intelligence'.

Erol, K., Hendler, J. & Nau, D. S. (1994), UMCP: A sound and complete procedure for hierarchical task-network planning, *in* 'International Conference on AI Planning Systems (AIPS)', pp. 249–254.

Erol, K., Nau, D. S. & Subrahmanian, V. S. (1995), 'Complexity, decidability and undecidability results for domain-independent planning', *Artificial Intelligence* **76**, 75–88.

Fikes, R. E. & Nilsson, N. J. (1971), 'STRIPS: a new approach to the application of theorem proving to problem solving', *Artificial Intelligence* **2**, 189–208.

Haigh, K. Z. & Veloso, M. M. (1998), 'Interleaving planning and robot execution for asynchronous user requests', *Autonomous Robots* **5**(1), 79–95.

Hebbar, K., Smith, S. J. J., Minis, I. & Nau, D. S. (1996), Plan-based evaluation of designs for microwave modules, *in* 'ASME Design Technical Conference'.

Kuter, U. & Nau, D. (2006), Controlled search over compact state representations, in nondeterministic planning domains and beyond, *in* 'National Conference on Artificial Intelligence (AAAI)'.

Kuter, U., Nau, D., Pistore, M. & Traverso, P. (2005), A hierarchical task-network planner based on symbolic model checking, *in* 'International Conference on Automated Planning and Scheduling (ICAPS)', pp. 300–309.

Kuter, U., Nau, D., Reisner, E. & Goldman, R. (2007), 'Conditionalization: Adapting forward-chaining planners to partially observable environments'. ICAPS 07 Workshop on Planning and Execution for Real-World Systems.

Muoz-Avila, H., Aha, D. W., Breslow, L. & Nau, D. (1999), Hicap: an interactive case-based planning architecture and its application to noncombatant evacuation operations, *in* 'AAAI/IAAI', pp. 870–875.

Nau, D., Cao, Y., Lotem, A. & Muoz-Avila, H. (1999), Shop: Simple hierarchical ordered planner, *in* 'International Joint Conference on Artificial Intelligence (IJCAI)', pp. 968–873.

Nau, D. S., Muoz-Avila, H., Cao, Y., Lotem, A. & Mitchell, S. (2001), Total-order planning with partially ordered subtasks, *in* 'International Joint Conference on Artificial Intelligence (IJCAI)'.

Smith, S. J. J., Nau, D. S. & Throop, T. A. (1996), 'A planning approach to declarer play in contract bridge', *Computational Intelligence* **12**, 106–130.

Wilkins, D. E. (1984), 'Domain-independent planning: representation and plan generation', *Artificial Intelligence* **22**, 269–301.

Wu, D., Parsia, B., Sirin, E., Hendler, J. & Nau, D. (2003), Automating daml-s web services composition using shop2, *in* 'International Semantic Web Conference (ISWC)'.