Frugal Hierarchical Task-Network Planning

by

Anthony C. Barrett

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1997

Approved by ______________________________

(Chairperson of Supervisory Committee)

____

____

Program Authorized
to Offer Degree ______________________________

Date ______________________________

University of Washington

Abstract

Frugal Hierarchical Task-Network Planning

by Anthony C. Barrett

Chairperson of Supervisory Committee: *Daniel S. Weld*

Department of Computer Science and Engineering

This thesis presents two planners that adapt tasks to attain goals...

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

$\{A_I\}_{I=0}M$: An action sequence [`start()`, $a_1, \ldots, a_m$, `stop()`].

$PRE(A)$: The precondition for executing an action a.

$EFF(A)$: The set of conditional effects for an action a.

Σ_ℓ^A: An extra precondition to assure that action a makes condition ℓ hold.

Π_ℓ^A: An extra precondition to assure that action a affect condition ℓ.

DNF(ρ,$\mathcal{I}$): A function to break an arbitrary Boolean equation ρ quantified over the initial conditions $\mathcal{I}$ into a disjunctive set of literal conjunct sets. conditions with the static literal ℓ. conditions with the static literal ℓ.

τ: A task.

$\tau_{\mathtt{S}}$: The distinguished problem task.

$\tau_N = (N{:}\alpha)$: A task with label n and type α

$\tau_0 = (0{:}\mathtt{do[START()]})$: The distinguished start task to begin the solution sequence.

$\tau_\infty = (\infty{:}\mathtt{do[STOP()]})$: The distinguished stop task to terminate the solution sequence.

$M \xrightarrow{\ell} N \in L$: An auxiliary constraint denoting a causal link or point protection depending on whether or not m denotes a task.

$\xrightarrow{\ell} N.I \in P_N$: A constraint defining how a CTN provides an externally requested condition.

$N.I \xrightarrow{\ell} \in E_N$: A constraint defining how a CTN uses an externally provided condition.

$E \overset{B}{\in} S$: A constraint such that there exists $e' \in S$ such that $e = e' \backslash b$.

$S^{\emptyset}$: Those elements in a set S not referring to universal variables.

S^B: Those elements in a set S that no longer have universal variables after a applying a substitution b.

$\mathcal{C}$: The set of constants mentioned in a problem

$\mathcal{G}(\mathcal{M})$: A power set of methods obtained by grounding methods in $\mathcal{M}$

$\mathcal{A}(\mathcal{M})$: Methods defining achieve tasks computed from $\mathcal{M}$

$\mathcal{U}(\mathcal{M})$: The set of methods obtained by expanding sets containing universal variables.

$\mathcal{G}^*(\mathcal{M})$: The set of methods for defining frugal decomposition.

$G(\mathcal{M})$: The grammar of methods defining task networks derivable from $\tau_{\mathtt{s}}$.

$\mathcal{L}(\mathcal{M})$: The set of primitive task networks derivable from $\tau_{\mathtt{s}}$.

$\langle N|Q\rangle$:

ACHIEVE TASK: a structure $(n{:}\mathtt{achieve}[\ell])$ denoting a literal ℓ to achieve.

CONTEXTUALIZED TASK NETWORK (CTN): a structure $\langle P : T, B, O, L : E\rangle$ denoting a task network and how it connects into a larger task network.

DECOMPOSITION METHOD: a structure $[\tau_n \Rightarrow \langle P_n : T_n, B_n, O_n, L_n : E_n\rangle]$ defining an abstract task in terms of its subtasks.

DERIVATION: a tuple $\langle T_*, S_*, D_*\rangle$ with components denoting methods for decomposing $\tau_{\mathtt{s}}$, a mapping from point protections to their corresponding tasks, and a set of unjustified tasks and effects.

FHTN PLANNING PROBLEM: a tuple $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho\rangle$ denoting a requirement to attain ρ from the initial conditions $\mathcal{I}$ by performing a procedure from $\mathcal{L}(\mathcal{M})$ using actions instantiated from $\mathcal{O}$.

OPERATOR: a structure $\langle f(v_1, \ldots, v_k), \mathsf{pre}, \mathsf{eff}\rangle$ defining actions with type f.

PERFORM TASK: a structure $(n{:}\mathtt{perform[p]})$ denoting an abstract procedure p to perform.

PRIMITIVE TASK: a structure $(n{:}\mathtt{do[a]})$ denoting an action a to execute.

TASK NETWORK: a tuple $\langle T, B, O, L\rangle$ with components denoting tasks binding constraints, ordering constraints, and auxiliary constraints respectively.

ACKNOWLEDGMENTS

To be determined

Chapter 1

INTRODUCTION

Functioning intelligently in a nontrivial world requires reasoning at varying levels of detail; people generally avoid thinking about trivia until after making the global decisions. For instance, when planning my daily trip to the office, I start by deciding whether to bicycle, drive, or take the bus. Notice that I did not start my deliberation by deciding which foot takes the first step; I do not consider trivial details like how to walk to the bus stop until after deciding to ride the bus.

Deliberating over a course of action at varying levels of detail is the hallmark of hierarchical task network (HTN) planning. An HTN planner initially develops plans with abstract tasks, like "ride bus", and later deliberates about which buses to ride. More precisely, an HTN planner represents a plan as a partially ordered set of tasks and the planning process incrementally replaces abstract tasks with progressively simpler sets of subtasks; each task either denotes an action or is defined in terms of its subtasks. For example, a task to go to the office can be defined in terms of subtasks to ride a bus; another set of possible subtasks would be those that drive a car. Ultimately, a task like "go to office" bottoms out in a partially ordered set of actions whose sequential execution gets me to work.

NOAH [Sacerdoti 75] introduced HTN planning by making two important innovations: the partial-order step representation and the definition of abstract tasks in terms of subtasks using decomposition schemata. Over the past twenty years, research influenced by NOAH has either focused on building "scruffy" planners to solve industrial sized planning problems (*e.g.*, NONLIN[Tate 77], O-PLAN[Currie & Tate 91], SIPE-2[Wilkins 90]) or on building "neat" planners that are provably sound and complete (*e.g.*, TWEAK[Chapman 87], SNLP[McAllester & Rosenblitt 91]).

Invariably, the "industrial strength" HTN planners gained most of their power through using abstract tasks and reduction, and the provably sound and complete planners dropped the notion of abstract tasks and built plans using the partial order step representation and adding one step at a time to attain a set of goals. This dichotomy apparently arose from the lack of a formal theory to characterize HTN planning problems and what plans constitute their solutions. While recent research on synthesizing plans one step at a time explored expressive action languages with conditional effects and universal quantification [Pednault 88, McDermott 91, Pednault 91, Penberthy & Weld 92], a theoretical formalization of HTN planning began to emerge [Yang 90, Kambhampati & Hendler 92, Erol et al. 94a].

These developments result in the realization that HTN planning's problems are strictly more expressive than partial-order (PO) planning's goal attainment problems in terms of specifying what action sequences constitute solutions, but goal attainment planners can reason with strictly more expressive action representations without sacrificing completeness. Also, prior to extending PO planners to handle more expressive actions, a PO planner was a special case of an HTN planner; thus HTN planning can be extended just as PO planning was extended to handle more expressive action representations.

In this thesis we combine HTN planning with actions having context sensitive effects to define frugal hierarchical task-network (FHTN) planning. When performing a task to achieve a specific goal, a frugal agent only executes those actions necessary for attaining the goal; similarly, an FHTN planner generates a sequence of actions to perform that part of a given task contributing toward goal attainment. Frugality's usefulness rapidly becomes apparent when considering actions and abstract tasks whose preconditions depend on where and how they are used. A task might be defined to satisfy all possible preconditions of an action, but those subtasks supporting unnecessary preconditions are extraneous and should be avoided.

1.1 Search Control in Planning

One common paradigm traditionally held by the planning community involves looking at planning as a search task. A planner typically starts with a problem and searches through action sequences until finding one that meets the problem's requirements. Prior to NOAH, these requirements were defined in terms of three components: what is initially true, what we wish to make true, and what we can do. For instance, I am currently at home, I want to be at office, and I have several ways to travel between two points. These components are respectively called the initial situation, the goal, and the actions.

Given a problem, the early planners built solutions by starting with empty action sequences and generating new sequences by appending actions to either the end or the beginning of a previously generated sequence. While the planners adding actions to the end deliberate over those actions performable after performing a sequence, the planners adding actions to the beginning deliberate over the actions needed to make the rest of the sequence performable to reach the goal. In either case, the deliberation often results in a choice from a set of appendable actions; thus a planner searches through a space of action sequences, where each arc corresponds to an addable action.

To illustrate the development of a search space, consider navigating the street map in figure 1.1 with the actions `N`, `S`, `E`, and `W`; these actions respectively travel North, South, East, and West to the next intersection. Supposing that the goal is to get to the office, a naive planner would start with the empty plan `[]` and generate sequences `[E]` and `[S]` that get to the office from one block away. The next step is to take one of these sequences, say `[S]`, and append a step to the beginning to get `[NS]`,`[ES]`, or `[SS]`. Planning continues to expand the space of sequences until one is found that navigates the map from home to the office. Since the number of actions addable to a particular plan is between 2 and 4, the number of possible n action plans is between 2^n and 4^n; thus our naive planner has a truly enormous search space to expand and explore.

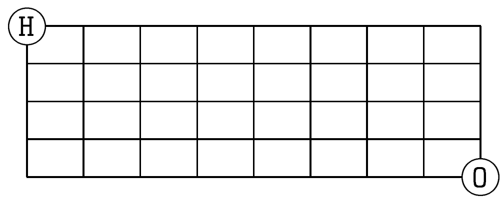


Figure 1.1: A street map marked with home `H` and the office `O`.

Since the number of action sequences in a search space can grow exponentially with the sequence length – or the distance from the empty plan, planning problems are generally difficult to solve; some form of search control is clearly needed to limit how much of the search space a planner computes. Although many search control techniques have been devised, they all fall into two categories: devise strategies for expanding the space or restructuring the space to make the solution closer to the empty plan.

1.1.1 Strategies for Expanding a Search Space

One way to control the search involves estimating the distance each generated sequence is from the solution, and build on the best sequence. For instance, a useful expansion strategy for our navigation problem would take all previously generated action sequences, note the intersections that they would start from to arrive at the office, and expand only that sequence starting from the intersection closest to home.

With a perfect estimate, a planner would always know which sequence to expand, and all problems become trivial. Unfortunately, a perfect estimate for all problems cannot exist; having one would imply a solution to some theoretically unsolvable problems, like determining if an arbitrary program will halt.

Since we cannot compute a metric like how far a sequence is from a solution, we devise heuristic ranking functions using weighted sums of metrics that we can compute and build on the sequence with the best rank. One such metric breaks up a problem's goal into parts and counts those parts that a particular sequence solves. By directing a planner's attention to those sequences solving the most parts, this metric's corresponding search control policy directs the planner to build on a sequence by solving one part at a time while preserving the solutions of previously solved parts. This is analogous to breaking a problem up, solving each piece separately, and combining the solutions. An example from our navigation problem might involve breaking the problem up in terms of getting on the correct street and the correct avenue; combining the two plans results in a plan to the correct intersection.

Korf [Korf 87] analyzed this metric and showed that its utility is problem dependent; sometimes the solutions to the parts interact so much that it hardly helps at all. For instance, a sequence solving all but one part of Rubik's cube may be farther from a total solution than a sequence that does not solve any part. We later showed [Barrett & Weld 94] that the success of PO planning is partially attributable to its making more problems amenable to being broken into parts. Instead of constructing sequences of actions, a PO planner constructs partially ordered sets of actions; thus a certain amount of interaction among all parts' solutions can be handled by interlacing while combining the actions solving each part.

As hinted in the previous paragraph, our research initially focused on search control in partial order planning. With the development of SNLP[McAllester & Rosenblitt 91], partial order planning became so well understood that various researchers focused on extending the action and goal formalisms. While context sensitive effects, metric effects, probabilistic effects, or sensory effects started appearing in actions, utility functions and metric quantities started appearing in goals. Even though the resulting planners were all provably sound and complete, each representational extension made the problem more computationally complex and the need for guiding a planner more critical.

Although PO planning enhances the utility of breaking problems into parts, solving each part may still be difficult. Not only are other search control methods used to overcome this difficulty, but they have found applications for ruling out obviously ill-advised plans. One such plan is illustrated in the following joke from the Marx Brothers [Davis 90].

> *Groucho: You say you're going to everyone in this house and ask them if they took the painting. Suppose nobody in the house took the painting.*
>
> *Chico: Go to the house next door.*
>
> *Grouch: That's great. Suppose there isn't any house next door.*
>
> *Chico: Well, then, of course, we got to build one.*
>
> *Animal Crackers*

While Chico's second statement sounds silly, we apply a certain amount of domain knowledge before laughing; to save a planner from deliberating over plans like this, we need to encode this domain knowledge to control the search. Since this domain knowledge can be arbitrarily complex, controlling a search with weighted sums becomes arbitrarily cumbersome and unintuitive.

One alternative way to encode search control knowledge involves explicitly defining it with search control rules. For instance, we can make a planner avoid Chico's failure by giving it a rule to avoid expanding any plan that built a house in order to interrogate its occupants. In the PRODIGY planner [Minton et al. 89] such a rule might look like figure 1.2, where the right-hand side (`RHS`) contains an action to perform when the test in the left-hand side (`LHS`) succeeds. In this case, the rule applies to nodes created by adding a `BUILD-HOUSE` action in order to satisfy a `(AT`

<x>) goal, and it deletes the nodes from the search space with a (REJECT NODE <node>) action.

```
(DO-NOT-BUILD-HOUSE-TO-GO-TO-IT
   (LHS (AND (CANDIDATE-NODE <node>)
             (CURRENT-OP <node> BUILD-HOUSE)
             (ON-GOAL-STACK <node> (AT <x>))))
   (RHS (REJECT NODE <node>))),
```

Figure 1.2: A PRODIGY search control rule that disallows plans that build houses in order to visit them

As shown in the previous example, rules can encode very local search control policies; our rejection rule only applies to a very small number of search space nodes. With ranking functions, the only way to encode such localized policies involved devising a metric for each localized policy. While this initially looks infeasible, we have shown [Barrett et al. 93] that ranking functions and heuristic control rules are not contradictory concepts; we can define localized metric functions with search control rules.

As Minton shows [Minton 90] a disadvantage with naively adding search control rules rapidly becomes apparent as the number of rules increases. Although rules reduce the number of nodes that a planner considers, each rule contributes to the amount of time spent on a node; thus a particular rule's utility depends both on how much it reduces the size of the search space and how much computational overhead it adds. In other words, just adding rules to reduce the search space is not good enough; we need to consider the cost of each rule and craft a set of rules that improves the planner's ultimate performance.

Another problem involves the locality of information accessible by a search control rule. Since search control rules can only look at the information in a particular node, it is difficult to hand generate a set of rules to make a planner exhibit a set of desired behaviors. When devising search control, people often think in terms of desired action sequences; reasoning about adding single actions is too detailed. Macros resolve part of this problem.

1.1.2 Strategies for Restructuring a Search Space

A *macro* is a sequence of actions that a planner uses just like a single action. For example, consider adding the macro `SSSSEEEEEEEE` to the office problem. With such a macro, our naive planner would start by generating the sequences `[S]`, `[E]`, and `[SSSSEEEEEEEE]`. At this point the planner stops because it detected a solution; executing the macro gets me to the office.

The idea of taking an action sequence and having a planner view it as a single macro action was first presented by Amarel[Amarel 68], where he notes that using macros can drastically reduce the size of a planner's search space; the end result is that finding the problem's solution requires very little search. In Amarel's example, a planner found a solution sequence of macros and then created an action sequence by replacing each macro with its defining actions; thus his planner alters the search space that starts with an empty plan and each arc adds a sequence of one or more actions corresponding to a macro. Given an appropriate set of macros, planning with an altered search space is easier because the distance to the solution is reduced.

Where search control rules attempt to improve a planner's performance by leading it on a path through the search space to a solution, macros attempt to encode the path directly. Unfortunately, just as perfect search control is impossible, so is encoding all paths; such an attempt would lead to a countably infinite number of arcs leading from the initial plan. Although it is theoretically possible to enumerate and test each arc, the end result is no faster than planning without macros; the enumeration looks just like the original search for a solution.

In general, just naively defining macros does not make the search space more tractable. Although it might shorten the distance to the solution plan, it also shortens the distance to a lot of other plans; adding one macro to our navigation example increased the number of intersections 2 steps away from home from 3 to 5, and the problem gets worse as the number of steps increases. Examples similar to this led Etzioni[Etzioni 92] to prove that just adding macros generally degrades performance.

In cases where macros were shown to help a planner [Amarel 68], they actually replaced the primitive actions; thus deleting actions is just as important as adding them with macros. Since macros replace single actions, care is needed to assure that the reformulated search space contains solutions whenever the original space does.

Although we have treated macros and search control rules as alternative methods to improve a planner's performance, planners can use both techniques simultaneously; in some cases the result is much better than the sum of the parts. For instance, Korf [Korf 87] showed how macros enhances the previously mentioned heuristic to count the pieces of a problem a particular sequence solves. In some cases this heuristic fails because extending a plan to solve another piece of a problem might momentarily require undoing part of an earlier piece's solution. Since the heuristic just counts the pieces done, it fails to recognize progress when the earlier piece's solution is undone. Macros can resolve this difficulty by encoding action sequences that undoes and redoes an earlier piece's solution; where adding each action separately runs into a problems half way through the sequence, adding the entire sequence in one step "bridges" the gap.

While search control rules and macros can improve a planner's performance, and weed out many obviously undesired solutions, there are some undesired solutions that they cannot weed out. For instance, suppose that I decide to cycle to work while planning my day; I had better plan on cycling back home at the end of the day. Since this behavioral requirement involves actions at both the beginning and end of my day's plan, I will call it a *global behavioral requirement*. While macros conveniently encode localized behaviors, and search control rules conveniently encode localized decision policies, neither conveniently encodes global planning decision policies.

One technique for conveniently encoding global behavioral requirements comes from HTN planning. By focusing on the reduction aspect of HTN planning, we see that task reduction methods in HTN planning are derived by extending on macros in two ways: any particular abstract action can have one *or* more reduction methods, and each method can use primitive *and* abstract actions. Where a macro defines an abstract action in terms of a single sequence of primitive actions, reduction methods define an abstract action in terms of a possibly infinite set of primitive action sequences. For instance consider a reduction method like:

$$\texttt{daily-plan} \Rightarrow \texttt{[cycle-to-office; work; cycle-home]},$$

where each of the three abstract subtasks has its own reductions. A method like this conveniently encodes my global requirement that I cycle home on days when I cycled to work.

Because reduction methods can conveniently encode global behavioral requirements we decided to incorporate them into UCPOP – a PO planner. As previously mentioned, PO planning resulted from ignoring reduction methods and focusing on adding actions to a partially ordered set; thus adding reduction methods to a PO planner turns it into an HTN planner. Since UCPOP is sound and complete, we need to prove that we can extend it into an HTN planner without sacrificing these theoretical properties; this requires extending the HTN planning formalism to handle actions containing universally-quantified conditional effects.

1.2 Extending HTN Planning

As previously mentioned, the HTN planning paradigm lacked a theoretical formalism to define what primitive action sequences solve an HTN planning problem; papers on the subject focused on informally describing systems that solved impressively large applied problems. These systems solved problems defined in terms of abstract tasks to perform in the context of an initial situation. Each was based on an algorithm that iteratively reduces abstract actions into networks of subactions and resolving interactions between actions using critic functions.

While a formalism has appeared[Erol et al. 94b], it mainly focuses on the early HTN planners like NOAH and NONLIN with primitive actions modeled using STRIPS. In this formalism every action has the same set of effects regardless of the state of the world when it executes. For instance, the template below uses the STRIPS notation to define the `N` action used to travel north to the next street on a street grid. As the definition shows, the `N` action always has two preconditions and two effects.

$$\begin{array}{lll}
\texttt{(defoperator} & \texttt{action}: & \texttt{N} \\
 & \texttt{precondition}: & \texttt{on-street}(\texttt{self}, u), \texttt{north-of}(u, v) \\
 & \texttt{effects}: & \texttt{on-street}(\texttt{self}, v), \neg\texttt{on-street}(\texttt{self}, u)
\end{array}$$

The limitations of the STRIPS action model become apparent upon noticing that actions with metric and conditional effects are starting to appear in HTN planners[Wilkins 92]; no formalism fully characterizes these planners or the problems they solve. Since UCPOP, the PO planner that we are adding reduction methods to, has an extended action representation based on Pednault's ADL[Pednault 89], we need to extend the current HTN planning formalism.

1.2.1 Achieving Preconditions vs. Reducing Tasks

Until this point, we have been focusing on planning as a search through a space of action sequences, but as HTN planning's iterative replacement characterization implies, the search is now through a space of task networks. In a task network, each node corresponds to a task, and each arc corresponds to a constraint between two tasks. For example, consider the search space in figure 1.3. The first task-network in the space has three sequential tasks, τ_1, τ_2, τ_3 to respectively cycle to the office, work, then cycle home. The other two task-networks are computed by reducing τ_2, the work task, into either two parallel tasks, τ_4 and τ_5, to spend my day writing a memo and preparing a talk or a single task, τ_6, to debug a program. The objective of the planning problem is to extend this search space until finding a task-network where all tasks denote executing primitive actions, and those actions can be ordered into a sequence.

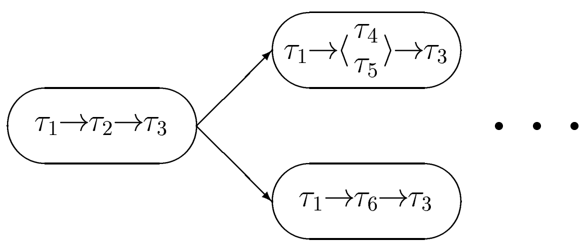


Figure 1.3: A graphical representation of an HTN planner's search space

While the early "refine action then resolve interactions" definition of an HTN planner is intuitively simple, problems start appearing when we start asking questions like, "what are interactions, and how do critic functions resolve them?" An even more basic question is, "How do constraints on an abstract action affect its subactions?" As these questions show, both phases of the iterative algorithm are ambiguously defined. These ambiguities result in a confusion over what action sequences solve an HTN planning problem. What does it mean for an action sequence to perform a task? Because of these questions, many researchers turned a blind eye toward reduction and created PO planning by focusing on adding primitive actions to plans with partially ordered sets of actions. Just like planning problems before NOAH, a PO planning problem is defined using an initial state, actions, and goal; any sequence whose execution attains the goal solves the problem.

To search for a solution, a PO planner starts with an initial task-network containing two tasks; we call them τ_0 and τ_∞, and they respectively denote the start and end of the plan. The objective is to add partially ordered actions between the initial tasks until executing any valid ordering of the actions results in attaining the problem's goals once τ_∞ is reached. To guide this process the PO planner determines the actions to add by keeping track of each existing action's preconditions.

To illustrate this process, consider figure 1.4, where the leftmost plan is the initial task-network. Supposing that the goals of the problem are to achieve some objective at work and return home, the next two plans in the search space are generated by either adding a task τ_1 or τ_2 to respectively enter the door to my apartment using my right or left foot. At this point the planner determines the preconditions to the action to enter my apartment.

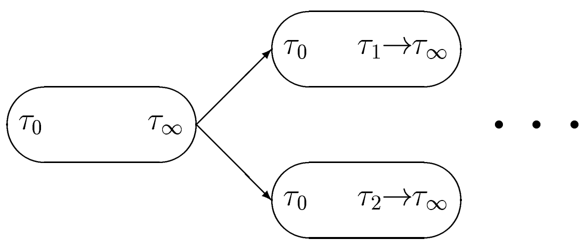


Figure 1.4: A graphical representation of a PO planner's search space

Of late all attempts to formalize HTN planning fall into two approaches: those that extends PO planning by adding abstract actions and reduction to a PO planner[Young et al. 94, Kambhampati 94], and those that give HTN planning an independent set of semantics[Erol et al. 94b Although we are extending a PO planner, we build on the latter approach. Our rational stems from the observation that a PO planning problem can be encoded as an HTN planning problem. The main idea behind this encoding is to take each action and compute a reduction method for each effect. For instance, our `N` action has two effects and would thus denote two encoded reduction methods; one would look like figure1.5

As we will see in chapter 2, not only does this encoding make both the HTN planner and PO planner have the same solution space, but it also makes a variant

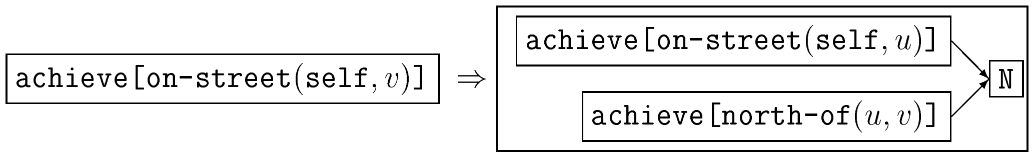


Figure 1.5: An encoded reduction method for the STRIPS operator N

of the HTN planning algorithm NONLIN equivalent to the PO planning algorithm SNLP; where SNLP added actions in accordance with the preconditions of existing actions, NONLIN added actions in accordance with achieve tasks originally derived from preconditions.

Since most research on extending action representations in PO planning involves extending the SNLP algorithm, noting the similarities between SNLP and NONLIN leads to natural extensions to the action representation in the HTN planning formalism. In this thesis the extensions involve universally quantified conditional effects.

1.2.2 Expressive Actions

To illustrate the utility of adding conditional effects to STRIPS, suppose I wanted to extend our representation of the N action to not only model my movement, but to also include the movement of anything that I might be holding. As illustrated in the following definition, such an action ends up having universally quantified conditional effects because I might hold any number of objects when I travel north. This action can be encoded as a set of actions in straight STRIPS, but the set would be exponentially large in the number of objects represented in the problem. This exponential comes from the fact that I would need a separate operator to characterize each possible set of objects that I might be holding during the move.

$$\begin{array}{lll}
\texttt{(defoperator} & \texttt{action:} & \texttt{N} \\
 & \texttt{precondition:} & \texttt{on-street}(\texttt{self}, u), \texttt{north-of}(u, v) \\
 & \texttt{effects:} & \texttt{on-street}(\texttt{self}, v), \neg\texttt{on-street}(\texttt{self}, u) \\
 & & \forall w\ \texttt{holding}(w)\ \textsf{causes}\ \texttt{on-street}(w, v) \\
 & & \forall w\ \texttt{holding}(w)\ \textsf{causes}\ \neg\texttt{on-street}(w, u)
\end{array}$$

Where PO planners using the originally encoded `N` action always kept track of two preconditions, UCPOP had to compute which preconditions to keep track of. For instance, if I wanted to transport a briefcase `B` north to a particular street, the `N` operator would have 3 preconditions: $\texttt{on-street}(\texttt{self}, u)$, $\texttt{north-of}(u, v)$, and $\texttt{holding}(\texttt{B})$. Like the original encoding for `N`, the first two determine that I can actually move to the specified street; the third depends on what I wish to move and must be computed.

In addition to computing preconditions when adding an action, UCPOP must also add preconditions to existing actions as planning progresses. There are two cases where an existing action can acquire new preconditions; the most obvious involves taking advantage of an existing step to satisfy extra preconditions. For instance, suppose that we decided to take both the briefcase `B` *and* an umbrella `U` north. UCPOP takes advantage of the existing `N` action to transport the umbrella too; it just adds the extra precondition $\texttt{holding}(\texttt{U})$. A less obvious case where actions acquire new preconditions involves assuring that an action does *not* have a particular effect; if we do not want to take a paycheck `P` to the office, we had better make sure that we are not holding it when travelling. With respect to our `N` action, this involves adding the precondition $\neg\texttt{holding}(\texttt{P})$.

Since UCPOP computes an ADL action's preconditions, we cannot directly encode ADL actions as HTN reduction methods. While one possible way is to first encode an ADL action as an exponential number of STRIPS actions and then encode each

STRIPS action, the resulting HTN problem has an intractably large number of task reduction methods. A preferable approach is to encode an ADL action using a task with the maximum possible number of subtasks and alter the HTN planning algorithm to only add those subtasks needed to make a task attain its required effects. This leads to FHTN planning; where UCPOP computes an action's preconditions from its required effects and ADL definitions, an FHTN planner computes a task's subtasks from its required effects and reduction method.

To provide a more precise characterization of FHTN planning, consider the reduction method in figure 1.6; it differs from figure 1.5's reduction method in two related ways: there are extra subtasks corresponding to the possible `hold`$(\mathbf{b})$ preconditions, and the variable **b** is universally quantified. Universally quantified variables in reduction methods is a notational convenience; each subtask labeled with such variables denotes a set of subtasks, where each element is computed by replacing the variable with a constant. Thus the **boldface** variables in our example imply that there are $3 + 2 \times |C|$ subtasks where $|C|$ is the number of problem specific constants.

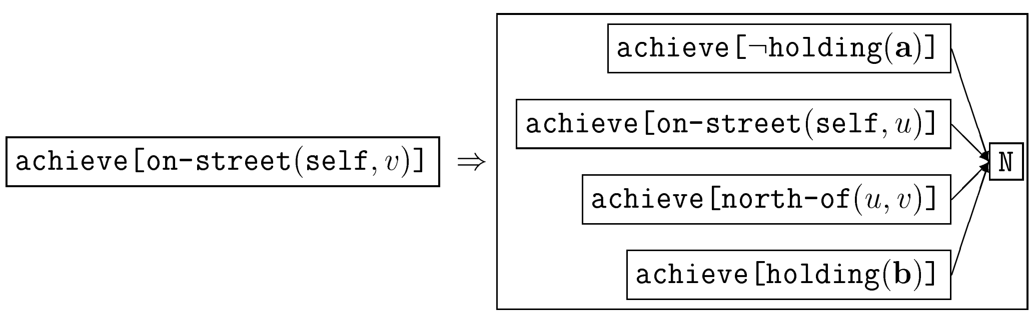


Figure 1.6: An encoded reduction method for the ADL operator `N`

Although each task has a potentially huge number of subtasks, the actual number used at any given time is considerably smaller; it depends on how the task is used. In the case of simply moving myself north, there are only 3 subtasks; since all I care about is my own movement, I can frugally delete all of the extraneous `holding` tasks.

1.3 Summary

This chapter introduces Frugal Hierarchical Task-Network (FHTN) planning by presenting its historical context, motivating hierarchical task networks (HTNs) by describing the limitations of other search control methods in planning, and describing how frugality facilitates adding HTN based search control to partial-order planners with context sensitive actions.

In chapter 2 we ground our work by discussing background planning results relevant to partial order planning and hierarchical task-network planning. These two versions of planning then combine in chapter 3 with our definition of frugal hierarchical task-network (FHTN) planning problems. Given our definition of FHTN problems chapter 4 shows how to extend a partial order planner into the FHTN planner UCPARSE. On a similar vein, chapter 5 shows how to extend a HTN planner into UCREDUCE, another FHTN planner.

Given our two FHTN planners, chapter 6 identifies common sequences of steps and subsequently proves each algorithm's soundness and completeness using properties of these sequences. Chapter 7 then empirically compares the performance of both algorithms, and determines that given appropriate flaw selection strategies each algorithm can outperform the other while solving the same problem; this leads to the definition of yet a third FHTN planner that degenerates into UCPARSE or UCREDUCE given the appropriate flaw selection strategy.

Finally, chapter 8 concludes this dissertation by comparing our algorithms with other existing HTN planners; these comparisons naturally lead into a discussion on avenues for future research.

Chapter 2

BACKGROUND

Since frugal hierarchical task-network planning draws on ideas from both hierarchical task-network (HTN) planning and partial-order (PO) planning, this chapter will discuss these planning paradigms and how they lead to our work. The main lesson is that although HTN planning problems are more expressive in terms of specifying acceptable solution action sequences, PO planning problems have more expressive action representations; merging the two paradigms for context sensitive actions leads to frugal HTN planning.

2.1 Common Planning Formalism

Before discussing HTN and PO planning, we must formulate their problems in a uniform framework. Both are similar in that they have three components: (1) a description of the world at a moment in time, (2) descriptions of the possible actions that an agent can perform, and (3) a description of the agent's desired behavior; all descriptions are made in a formal language. For example, for STRIPS planning problems the language is:

Definition 1 *A* STRIPS ACTION *consists of an action name, a* PRECONDITION LIST*, and an* EFFECT LIST*. The elements of the precondition and effect lists are all function-free, atomic expressions. A* STRIPS PLANNING PROBLEM *is a tuple* $\langle \mathcal{O}, \mathcal{I}, \Omega \rangle$ *with three components:* $\mathcal{O}$ *denotes a set of STRIPS operators,* $\mathcal{I}$ *denotes a complete set of initial propositions, and* Ω *denotes a set of goal propositions.*

2.1.1 Actions and Situations

All classical planners describe actions and situations with some variant of the STRIPS formalism, where the initial world description is completely described by a set $\mathcal{I}$ of ground atomic literals, called a *situation*. Commonly, all literals explicitly represented in a situation are positive, and the absence of a positive literal implies the presence of its negative counterpart; this implication is called the *closed-world assumption*[Reiter 80]. For instance, a *Blocks World* situation involving a three block tower on a `Table` is completely described with the set:

$$\{\texttt{clear}(\texttt{A}), \texttt{on}(\texttt{A}, \texttt{B}), \texttt{on}(\texttt{B}, \texttt{C}), \texttt{on}(\texttt{C}, \texttt{Table}), \texttt{clear}(\texttt{Table})\}.$$

As a situation describes a state of the world, actions to change the world are described as directives to add and remove literals from a situation. When an action adds its effects to a situation, their negated counterparts are removed. Due to the implicit presence of negated literals in a situation, the additions and removals of literals are either implicit or explicit depending respectively on whether of not the literal is negative.

Since an action cannot be executed in all situations, it is ultimately described by two sets of grounded literals: the literals to add/delete and the preconditions; an action can be performed only in those situations containing its preconditions. Sets of actions are commonly described using variablized operators that also include a set of codesignation (and noncodesignation) constraints[1]. For instance, the following operator describes actions that move any exposed block u onto the table:

---

[1] This representation is in fact a simplification of that used by STRIPS [Davis 90, Fikes & Nilsson 71].

$$
\begin{array}{lll}
\texttt{(defoperator} & \texttt{action}: & \texttt{unstack}(u) \\
 & \texttt{precondition}: & \{\texttt{on}(u,v), \texttt{clear}(u)\} \\
 & \texttt{effects}: & \{\texttt{on}(u,\texttt{Table}), \texttt{clear}(v), \neg\texttt{on}(u,v)\} \\
 & \texttt{equals}: & \{(u \neq v), (u \neq \texttt{Table}), (v \neq \texttt{Table})\}\texttt{)}
\end{array}
$$

In all cases the solution to a problem is a sequence of actions that conforms with the desired behavior and is executable from the initial world description, but PO and HTN planners differ regarding how they describe desired behaviors. In PO planners, this component is defined using a goal that must be attained when the agent finishes the sequence, but HTN planners represent this component using a coordinated set of tasks that the agent performs by executing the sequence. Regardless of how the behavior is defined, all planners inherit the following implicit assumptions from the STRIPS formalism:

- **Deterministic Context-Free Effects.** For any given situation, an action always has the same effect.

- **Static World.** The only way to change a situation is to explicitly do an action.

- **Atomic Actions.** Actions are atomic transformations from one situation to the next. No two actions can overlap and they cannot be partially done.

- **Full Knowledge.** An action's effects are fully known because the action is fully understood, and the current situation is fully known.

Prior to Sacerdoti's NOAH [Sacerdoti 75], classical planning was viewed as a search through a graph of situations — *i.e.*, a graph in which nodes are labeled with a set of propositions that specify what is true in that state of the world. A STRIPS planning problem can be solved by searching through such a graph. $\mathcal{I}$ specifies the initial situation, Ω specifies a set of goal situations, and the operators in $\mathcal{O}$ specify actions that correspond to the directed edges. If an action's preconditions are satisfied in a situation, then an edge leads from that node to the node denoting the effect of applying that action. The purpose of the search is to find a path from the initial situation to a goal situation. The solution to the planning problem consists of the actions associated with the edges of this path. An example of such a search space appears in figure 2.1.

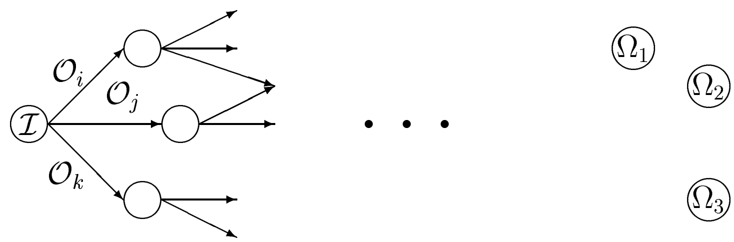


Figure 2.1: A STRIPS planning problem $\langle \mathcal{O}, \mathcal{I}, \Omega \rangle$ can be solved by searching through a graph of situations. The objective is to find a path from $\mathcal{I}$ to a situation labeled with $\Omega_i \supseteq \mathcal{I}$. Edges from a situation correspond to operators that can be performed in that situation.

2.1.2 Task-Networks

One of the major contributions of NOAH was a conceptual shift: instead of viewing planning as search through a space of situations, NOAH searched through a space of (possibly incomplete) plans. All PO and HTN planners perform similar searches, but the two planning paradigms may differ with respect to how they formulate the topology of the search space. In all cases the plans are denoted by task-networks.

Definition 2 *A* task network *as a tuple:* $\langle T, B, O, L \rangle$ *in which* T *denotes a set of tasks;* B *denotes a set of binding constraints over the variables mentioned by tasks in*

T; O denotes a set of ordering constraints that specify a (possibly partial) order on T, and L denotes a set of auxiliary constraints.

A task-network's constraints are used to coordinate its tasks, and each element $\tau_n \in T$ has the form $(n{:}\alpha)$, where n and α are respectively a unique label and the task's type. The labels are necessary to assure that constraints correctly refer to their tasks. While the constraints in B refer to constants or variables in tasks, those in O and L refer to the tasks themselves. Figure 2.2 illustrates a task-network for demolishing the previously described 3 block tower using tasks to execute the previously described `unstack` operator.

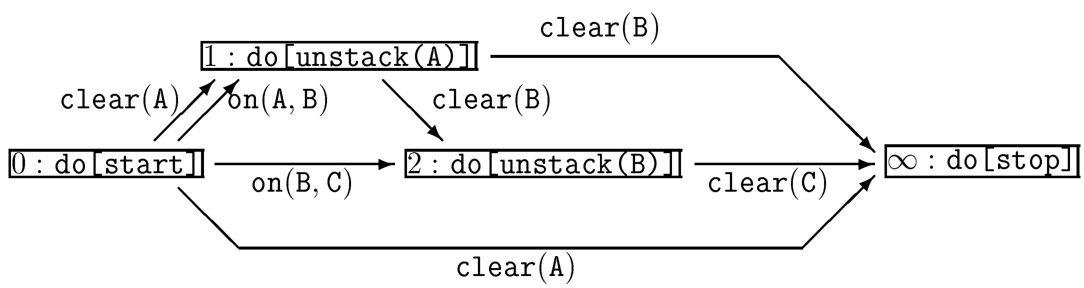


Figure 2.2: A plan for flattening a 3 block tower with `A` on `B` and `B` on `C`.

Although the two planning paradigms differ with respect to how they formulate their search spaces, the search space is always defined by both the planning problem and the planner, and the objective is to find a solution node. For PO planning such a node contains a task network like the one in figure 2.2. As our example illustrates, in a *solution task-network* all tasks denote actions, and each action's preconditions are accounted for by auxiliary constraints. If the actions can be serialized such that all constraints are satisfied, the serialization forms a action sequence that solves the problem.

In addition to the task-ordering and variable-binding constraints, there are commonly two types of auxiliary constraints: protection intervals and point protections.

A *protection interval* $n \xrightarrow{p} m$ denotes that proposition p remains true between tasks τ_n and τ_m. As illustrated in our example, these intervals are commonly used to denote the causal dependencies between tasks. Essentially, the first task makes some proposition true in order to satisfy a preconditions of tasks that occur during the interval. The proposition's persistence during the protection interval is required to preserve this functional relationship between tasks.

Sometimes a task has a precondition p, but does not occur during an interval that guarantees p's truth; point protections handle these cases. A *point protection* $\langle p@n \rangle$ is a constraint requiring that p is true before it is needed by task τ_n. Since no source action is specified, point protections are satisfied by adding ordering constraints to ensure that p is true when needed by τ_n. Often no constraints will be needed because the precondition must be true; in our example, this occurs for the `clear(A)` precondition of task (1:`do[unstack(A)]`).

2.2 HTN Planning

In HTN planning there are two types of tasks: primitive tasks denoting actions to execute and non-primitive tasks denoting procedures to perform. The problem involves taking a specified coordinated set of tasks and finding an action sequence to perform them. NOAH[Sacerdoti 75], the first HTN planner, takes the coordinated set of tasks, and uses decomposition methods to reduce non-primitive tasks by substituting them with coordinated sets of subtasks until only primitive tasks remain. When tasks interfere with each other, critic functions perform arbitrary transformation repairs; such a repair can be as simple as adding a coordinating constraint or as complex as modifying sections of the task-network. One of NOAH's flaws was that it did not backtrack when no critic function could resolve an unwanted interaction, but this problem was quickly repaired in the next HTN planner NONLIN[Tate 77].

2.2.1 Decomposition Formalism

One of HTN planning's initial difficulties was its emergence as an informally defined technique; only recently has a formal characterization of HTN planning problems begun to appear [Erol et al. 94b, Kambhampati & Hendler 92, Yang 90]. In the three component formalization of a planning problem, HTN planning inherits the STRIPS approach toward representing actions and situations, but the behavioral requirement is a coordinated set of tasks to perform. While performing a primitive task like (1:`do[do[unstack(A)]]`) is simply a matter of executing the action, more abstract *non-primitive* tasks correspond to multi-action procedures; thus the behavior has five components: T, B, O, L, and $\mathcal{M}$. Of these components the first four denote the requested coordinated set of tasks, and $\mathcal{M}$ is a set of decomposition methods for defining non-primitive tasks in terms of their subtasks.

There are two types of non-primitive tasks: `perform[`t`]` is a directive to perform some compound procedure t, and `achieve[`p`]` is a directive to attain a proposition p. In both cases, each abstract task corresponds to a (possibly infinite) set of primitive task-networks, where all tasks in a *primitive task-network* $\langle T, B, O, L\rangle$ denote actions to execute. These task-networks are recursively defined by elements of $\mathcal{M}$, where each element has the form:

$$\tau_n \Rightarrow \langle T_n, B_n, O_n, L_n\rangle.$$

For example, figure 2.3 illustrates a non-primitive task (n:`achieve[`$clear$`(v)]`) defined in terms of two subtasks coordinated by a causal link and a point protection. The resulting task clears a block v by recursively clearing the block u above and then unstacking.

The point where compound procedures and attainment tasks differ involves the possibility that a task to attain a proposition p can be constrained to occur when p is already true. In these cases, the appropriate behavior might be to wait for p to become true. To correctly model this, each attainment task to `achieve[`p`]` has a

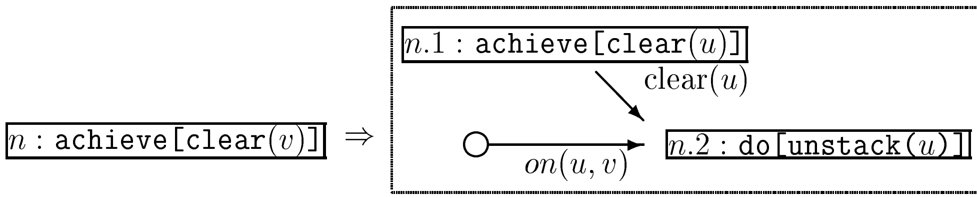


Figure 2.3: Graphical representation of a decomposition method defining tasks with type `achieve[clear(`v`)]`.

corresponding *phantomizing* decomposition:

$$(n{:}\texttt{achieve[p]}) \Rightarrow \langle\{(n.1{:}\texttt{do[no-op]})\}, \emptyset, \emptyset, \{\langle p@n.1\rangle\}\rangle.$$

Ultimately, decomposition methods define a non-primitive task as a set of primitive task-networks, where each primitive task-network is a coordinated cross product of primitive task-networks for a method's subtasks. For instance, the phantomization method to `achieve[clear(`v`)]` defined a single primitive task-network; the method in figure 2.3 takes this initial task-network and recursively defines an infinite set of task-networks, thus an `achieve[clear(`v`)]` task is a set of chains of primitive `do[unstack(`u_i`)]` tasks.

Similarly, a solution to an HTN planning problem $\langle \mathcal{O}, \mathcal{I}, tn, \mathcal{M}\rangle$ is another coordinated cross product of primitive task-networks corresponding to the tasks in $tn = \langle T, B, O, L\rangle$. For instance, suppose that the problem is to clear a set of blocks. The appropriate primitive task-network to clear a single block v depends on the number of blocks above and is defined by the `achieve[clear(`v`)]` task. The solution to the problem is obtained by taking a task-networks for each block to be cleared and combining them.

2.2.2 Generic HTN Planner

Since NOAH introduced HTN planning, a number of HTN planners have been devised [Currie & Tate 91, Tate et al. 94, Erol et al. 94b, Tate 77, Wilkins 88, Wilkins 90], but they all use slight variants of NOAH's basic underlying algorithm. There have been several papers formalizing this algorithm [Yang 90, Kambhampati & Hendler 92, Erol et al. 94b]; we describe it using figure 2.4. Since each of *HTN-PLAN*'s arguments matches a symbol in the HTN problem formulation $\langle \mathcal{O}, \mathcal{I}, \langle T, B, O, L \rangle, \mathcal{M} \rangle$, the initial invocation is obvious.

Algorithm: *HTN-PLAN*$(\langle T, B, O, L \rangle, \mathcal{O}, \mathcal{M}, \mathcal{I})$

1. **Termination:** If T contains no non-primitive tasks, then backtrack when $comp(\langle T, B, O, L \rangle, \mathcal{O}, \mathcal{I}) = \emptyset$; otherwise return an element of it.
2. **Task selection:** Let $\tau = (n : \alpha)$ be a non-primitive task in T.
3. **Reduction:** Let $d_n \in \mathcal{M}$ be a decomposition method for tasks of type α and let $\langle T', B', O', L' \rangle = reduce(\langle T, B, O, L \rangle, \tau, d_n)$. *Backtrack point: Each method for decomposing τ must be considered for completeness.*
4. **Interaction resolution:** Let $\langle T'', B'', O'', L'' \rangle$ be one of the task-networks returned by $critic(\langle T', B', O', L' \rangle, \mathcal{O}, \mathcal{I})$. *Backtrack point: Each member of the returned set must be considered for completeness.*
5. **Recursive invocation:** *HTN-PLAN*$(\langle T'', B'', O'', L'' \rangle, \mathcal{O}, \mathcal{M}, \mathcal{I})$.

Figure 2.4: Basic HTN planning algorithm

During planning, the termination step takes a primitive task-network and attempts to compute a problem's solution action sequence; it uses the *comp*() function to find a ground linearization of T's actions that satisfies all constraints. This step fails and the planner backtracks when no such sequence exists. If the task-network contains non-primitive tasks, the task and schema selection steps arbitrarily pick[2] a non-primitive task τ_n and reduce it with a nondeterministically chosen a method in $\mathcal{M}$; the chosen method defines task τ_n in terms of a plan fragment $\langle T_n, B_n, O_n, L_n \rangle$.

Using a method to reduce the non-primitive task τ_n in $\langle T, B, O, L \rangle$ involves substituting τ_n with the plan fragment and redirecting constraints from τ_n to the fragment's subtasks. The result would be a new plan:

$$\langle (T - \tau_n) \cup T_n, B \cup B_n, O' \cup O_n, L' \cup L_n \rangle,$$

where O' and L' are computed by the redirection. In general, the HTN planners differ with respect to how they redirect constraints. The simplest approach toward redirection involves adding tasks $\tau_{n.0}$ and $\tau_{n.\infty}$ that respectively execute phantom `no-op` actions before and after the fragment and redirecting constraints toward them; for instance, each ordering constraint of the form $\tau_n \prec \tau'$ is replaced with a constraint $\tau_{n.\infty} \prec \tau'$.

After each reduction, or phantomization, the function *critic*() is called to resolve interactions. In general, this function consists of a set of critics, where a *critic* is a test for an interaction coupled with a set of repair transformations. The return value is a set of task-networks due to the fact that more than one interaction might be detected, and each interaction might have more than one possible repair. When no interactions are detected, the return value is a singleton set containing the original task-network.

There are two classes of interactions: constructive and destructive. While constructive interactions typically involve point protections, destructive interactions typically involve threats to protection intervals. For instance, one way to assure the satisfaction of a point protection $\langle p@n \rangle$ is to identify a task τ_m that asserts p and replace the point protection with a protection interval $m \xrightarrow{p} n$; thus task τ_m construc-

[2] Task selection is *deterministic*; it may affect the algorithm's performance, but it has no effect on completeness.

tively interacts with τ_n. On the destructive side, we say that a protection interval $n \xrightarrow{p} m$ is *threatened* if some task τ_l may possibly be ordered between τ_n and τ_m, and τ_l negates a proposition that possibly unifies[3] with p. There are three ways to repair a threat: (1) order τ_l before τ_n, (2) order τ_l after τ_m, and (3) add variable-binding constraints to assure that τ_l cannot delete a proposition that can unify with p.

Although all planners eventually resolve all threats using the same set of repairs, the point where each threat is resolved depends on the planner. While in early resolution the critic detects the threat regardless of the tasks involved, in late resolution the threat is detected when all tasks are primitive. Early resolution provides a computational advantage by quickly detecting a threat that cannot be repaired without making the task-network's constraints contradictory, but as Yang[Yang 90] points out, the problem with early detection is that it can lead to incompleteness; there are cases where a threat involving abstract tasks cannot be fixed with one of the three repairs, but the threat becomes repairable once the tasks are reduced.

In general the critic function's definition varies with the planner and causes a lot of difficulty when trying to formalize HTN planning. In all cases, the set of critics is responsible for detecting interactions and repairing the task-network, but planners differ with respect to how the critics are defined. In some cases, the same planner can differ from domain to domain when domain specific critics are suggested. Although the generality of critics are a major source of informality, they are still very useful; they are so useful that the planner COLLAGE[Lan 95] was developed by dropping reduction and defining domains in terms a set of critic functions alone. In those domains where COLLAGE uses reduction, it is introduced as a critic function.

---

[3] Two propositions possibly unify if they can be unified by adding variable-binding constraints to a task-network without making that task-network's variable constraint set B inconsistent.

2.3 PO Planning

Although the informal development of HTN planning led to powerful planners that could find solutions to large problems, it also led to conceptual difficulties when trying to prove that an HTN planner will always find a solution when one exists. Completeness arguments require formal definitions of the planning algorithm and a problem's solution space; until recently, HTN planning lacked such formal definitions. To combat these difficulties, Chapman [Chapman 87] adopted the STRIPS problem definition with its formally defined solution space; He then started research on partial order planning by ignoring the decomposition feature of HTN planning while keeping the partial order step representation. The resultant formally defined PO planning algorithm, called TWEAK, is provably sound and complete.

Unlike HTN planners that build plans by substituting a task with its subtasks, PO planners synthesize plans by adding one action at a time. They still use a critic function, but this function has a formal definition that leads to the planner's correctness and completeness. In terms of our search through a space of task-networks, a planning problem is encoded in an *initial task-network* $\langle T, B, O, L \rangle$ consisting of two tasks $\tau_0 = (0 : \texttt{do[start]})$ and $\tau_\infty = (\infty : \texttt{do[stop]})$. The task τ_0 adds $\mathcal{I}$, and τ_∞ has Ω for preconditions. This task-network has no variable-binding or auxiliary constraints, but O has one constraint to force τ_0 before τ_∞. The set $\mathcal{O}$ defines what can be added to the set T in a task-network. The goal of the search is to find a task-network where the preconditions of all actions are necessarily true.

2.3.1 SNLP

One problem with TWEAK was the conceptual complexity of its critic function. McAllester[McAllester & Rosenblitt 91] later advanced PO planning by devising a simpler algorithm, called SNLP and proving it sound, complete, and systematic. Using the planning as search paradigm, systematicity means that an algorithm is guaranteed to search among task-networks in an irredundant fashion – visiting every possible task-network exactly once. This property is reflected in the structure of each planner's search space in that the directed edges form a tree rooted in the initial task-network.

Like some previous planners (*e.g.*[Kambhampati & Hendler 92, Tate 77, Warren 74, Yang & Tenenberg 90]) but unlike TWEAK, McAllester's algorithm uses *causal links* to record the purpose for introducing a task into a plan and to protect that purpose. If a task τ_n adds a proposition p to satisfy a precondition of task τ_m, then $n \xrightarrow{p} m$ denotes the causal link; this differs from a protection interval in that it requires that p is added by τ_n and is a precondition of τ_m. McAllester's key innovation is a clever, methodical technique for creating and protecting causal links.

As illustrated in figure 2.5, SNLP bears a remarkable resemblance to the algorithm description of a generic HTN planner; the differences are that reducing abstract tasks is replaced with selecting operators and the critic function is formalized. The result is a planner that builds a task-network by adding one action at a time. Not only does this algorithm guarantee soundness and completeness, but all plans generated are *frugal* because each added action contributes toward achieving the ultimate goal and causal links guarantee the protection of that purpose.

Each precondition p of a task $\tau_m \in T$ is an *open condition* if it has no corresponding causal link $n \xrightarrow{p} m \in L$. The algorithm searches for a solution task-network $\langle T, B, O, L \rangle$ by eliminating open conditions in *agenda* while ensuring the safety of causal links in L. In the initial invocation of SNLP, $\langle T, B, O, L \rangle$ is the initial plan-state, and *agenda* has the set of preconditions of τ_∞

Our using least commitment for variable bindings has a subtle effect on the causal-link-protection step. In the absence of unbound variables, protecting a causal link $n \xrightarrow{p} m$ from a task τ_l simply involved ordering τ_l before τ_n or after τ_m. With the introduction of unbound variables we get the extra possibility of adding variable constraints between the effects of τ_l and the proposition p. For example, there are 4 different sets of constraints that can be added to protect $n \xrightarrow{\mathtt{on}(a,b)} m$ from a task τ_l that

Algorithm: *SNLP*$(\langle T, B, O, L\rangle, agenda)$

1. **Termination:** If *agenda* is empty, report success and stop.
2. **Goal selection:** Let q be a proposition in *agenda*, and let τ_m be the task for which q is a precondition.
3. **Operator selection:** Let τ_n be a task that adds q (either a new or existing task possibly prior to τ_m). If no such task exists then backtrack. Let $L' = L \cup \{n \xrightarrow{q} m\}$, $T' = T \cup \{\tau_n\}$, $O' = O \cup \{\tau_n \prec \tau_m\}$, and $B' = B \cup$ the set of variable bindings to make τ_n add q. *Backtrack point: Each existing and possibly addable task must be considered for completeness.*
4. **Update goal set:** Let $agenda' = (agenda - \{\langle q@m\rangle\}) \cup$ preconditions of τ_n, if new.
5. **Causal link protection:** A task τ_l *threatens* a causal link $n \xrightarrow{p} m$ when it occurs between τ_n and τ_m and it adds or deletes p. For every task τ_l that might threaten some causal link $n \xrightarrow{p} m \in L'$:
   - Ensure that τ_l does not threaten $n \xrightarrow{p} m$ by adding constraints to O' and/or B'. *Backtrack point: Each way to protect* $n \xrightarrow{p} m$ *from* τ_l *must be considered for completeness.*
6. **Recursive invocation:** *SNLP*$(\langle T', B', O', L'\rangle, agenda')$.

Figure 2.5: The Systematic Non-Linear Planner (SNLP)

deletes $\mathtt{on}(x, y)$.

1. $\{\tau_l$ before $\tau_n\}$,

2. $\{\tau_l$ after $\tau_m\}$,

3. $\{\tau_l$ between τ_n and τ_m, $x \neq a\}$,

4. $\{\tau_l$ between τ_n and τ_m, $x = a$, $y \neq b\}$

The addition of codesignation as well as noncodesignation constraints is required to ensure systematicity. In general, the number of ways to protect a causal link is linear in the number of unbound variables involved.

2.3.2 UCPOP

One of the limitations with the STRIPS formalism is that it is not expressive enough to represent many actions. For instance, consider letting go of a glass. If it rests securely on another object, it stays there; otherwise, it falls and possibly breaks. Admittedly, ungrasping can be modeled with three STRIPS operators for: supported objects, unsupported fragile objects, and other unsupported objects. Modeling actions like this becomes problematic when considering actions like carrying a briefcase from home to the office. Not only does the briefcase move, but so does each item inside. This activity would require a STRIPS action for each possible set of objects.

As our examples suggest, modeling an action's effects with a set of literals makes an over restrictive assumption that actions always have the same effects. One approach to representing actions that circumvents this limitation comes form Pednault's Action Description Language (ADL)[Pednault 86a]. As in STRIPS, each action has a name, a precondition, and a set of effects. The difference is that the precondition is represented using a formula, and each effect can have an *extra* precondition associated with it. Not only are effects conditional, but they can also be universally quantified. For example, the carry(*from,to*) operator below is a template defining actions that carry a briefcase b, with its contents, to some location. The precondition is simply that the briefcase is at an old location, but the effects depend on what is in the briefcase. All of the contained objects get moved during the action.

$$
\begin{array}{lll}
\texttt{(defoperator} & \texttt{action}: & \texttt{carry}(b, from, to) \\
 & \texttt{precondition}: & \texttt{at}(b, from) \\
 & \texttt{effects}: & \texttt{at}(b, to), \neg\texttt{at}(b, from), \\
 & & \forall u\ \texttt{in}(u, b) \wedge (u \neq b)\ \textsf{causes}\ \texttt{at}(u, to) \\
 & & \forall u\ \texttt{in}(u, b) \wedge (u \neq b)\ \textsf{causes}\ \neg\texttt{at}(u, from)
\end{array}
$$

In order to plan with context-sensitive actions, UCPOP[Penberthy & Weld 92] was created by modifying the SNLP algorithm. The changes involved altering how SNLP maintained its set of open goals. Where SNLP only adds open goals when inserting new steps into a plan, UCPOP also adds open goals to assure or disable each existing steps' conditional effects. UCPOP disables an effect upon determining that it threatens some causal link; assuring an effect happens when it is used to satisfy some other open goal. While assuring an effect involves adding the effect's precondition to the open goals, disabling requires negating the precondition before adding it to the open goals. Thus, as more of a step's effects become relevant to a planning problem, more open goals are added.

Another representational extension supported by UCPOP involves goals and preconditions using formulae with bounded quantification. For example, the formula "$\forall[u : \texttt{book}(u)]\neg\texttt{at}(u, \texttt{Office})$" denotes a requirement that there are no books at the office. In order to implement this formula, UCPOP assumes that the bounding predicate book does not appear in any step's effects. The example is handled by taking each ground literal in the initial situation with the predicate book, computing a formula by substituting the literal's argument for u in $\texttt{at}(u, \texttt{Office})$, and adding the result to the open goals.

2.4 *Bridging the Gap*

Despite HTN planning's informal characterization, HTN planners have been successfully applied to problems in industry[Russell & Norvig 95]. Curiously, even though PO planners have formal characterizations and completeness proofs, they have yet to be successfully applied. According to Drummond[Drummond 94], the three capabilities apparently responsible for the success of HTN planners are: (1) explicit languages for documenting a plan's causal structure, (2) hierarchical operator expansion, and (3) simple propositional resource allocation. While *Simple propositional resource allocation* involves the declaration/protection of causal links and has been featured in PO planning since SNLP, the first two capabilities are exclusively provided by HTN planners. The final result is that (1) confers increased control over the acceptable solutions and (2) confers increased control over the planning process.

In order to bridge the gap and apply formal results from PO planning to HTN planning, we start by illustrating a technique for converting STRIPS problems into HTN problems. One important property of a converted STRIPS problem is that it can make an HTN planner *simulate* SNLP. This simulation implies that formal results in PO planning also apply to some classes of HTN planning problems. Also, since recent advances in PO planning for more expressive action representations resulted from extending the SNLP algorithm, determining how to similarly extend the simulation results in comparable advances to HTN planning.

2.4.1 Simulating SNLP

Initially, PO planning was a special case of HTN planning; this follows from the fact that a STRIPS problem $\langle \mathcal{O}, \mathcal{I}, \Omega \rangle$ can be converted into an HTN problem. While the first two components of a problem remain unchanged, the behavioral component must be computed. The conversion involves taking the goals Ω and computing the HTN problem's initial task-network $\langle T, B, O, L \rangle$; the process is similar to creating a PO planner's initial task-network, but also involves taking each goal in Ω and creating an achieve task. For instance, the initial task network for a STRIPS problem to clear a three block tower is illustrated in figure 2.6.

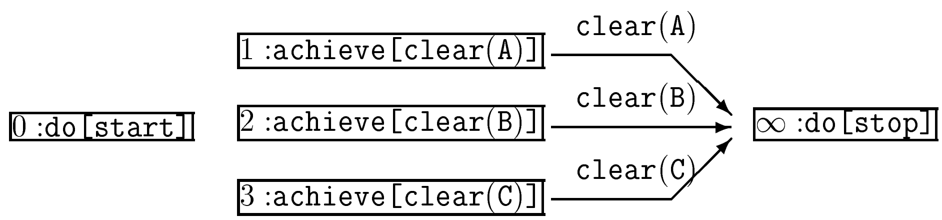


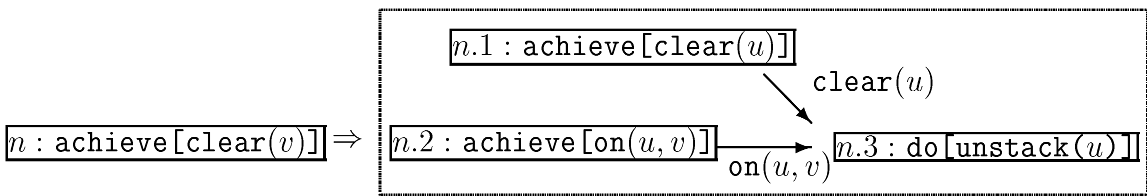
Figure 2.6: Initial task-network for clearing 3 blocks

Computing an achieve task's decomposition methods involves taking each operator $\mathcal{O}_i \in \mathcal{O}$ that asserts the achieve task's goal p and computing a decomposition method for `achieve[`p`]` with a primitive $(n :$ `do[`$\mathcal{O}_i$`]`$)$ and achieve subtasks for all of $\mathcal{O}_i$'s preconditions. Figure 2.7 illustrates an example of this computation where task $(n :$ `achieve[`$clear(v)$`]`$)$ uses an effect of the `unstack(`u`)` action. One feature not illustrated in the figure is the requirement that the `equals:` entries are reflected in the binding constraints of the decomposition method's plan fragment $\langle T_n, B_n, O_n, L_n \rangle$.

With the appropriate critic function, the converted STRIPS problem will make an HTN planner *simulate* SNLP. Where SNLP solves a problem by iteratively selecting an open goal, satisfying it, and protecting causal links, an HTN planner solves a problem by iteratively selecting a non-primitive task, reducing it, and resolving interactions; therefore, with a converted STRIPS problem, SNLP's reference to an open goal is matched by an HTN planner's reference to the associated achieve task. Actually, reduction does not exactly simulate open goal satisfaction in SNLP; where reduction phantomizes a task by adding a point protection, SNLP satisfies a goal by linking to an existing step. This discrepancy can be can be repaired when the HTN planner handles interaction by calling its critic function; thus the critic function consists of two steps: (1) replace each point protection $\langle p@n \rangle$ with a causal link $m \xrightarrow{p} n$ from a

$$
\begin{array}{lll}
\texttt{(defoperator} & \texttt{action}: & \texttt{unstack}(u) \\
 & \texttt{precondition}: & \{\texttt{on}(u,v), \texttt{clear}(u)\} \\
 & \texttt{effects}: & \{\texttt{on}(u,\texttt{Table}), \texttt{clear}(v), \neg\texttt{on}(u,v)\} \\
 & \texttt{equals}: & \{(u \neq v), (u \neq \texttt{Table}), (v \neq \texttt{Table})\}\texttt{)}
\end{array}
$$

(a)



(b)

Figure 2.7: A decomposition method computed from `unstack(`x`)`

nondeterministically chosen existing primitive task τ_m adding p, then (2) resolve each threat not involving achieve tasks.

2.4.2 Causal Structure Languages

By *explicit languages for documenting a plan's causal structure*, Drummond refers to the ability to explicitly control how to satisfy an action's preconditions. In HTN planning, new tasks and actions are added while replacing a non-primitive task with a decomposition method's task-network; thus how to satisfy an action's preconditions is defined by the auxiliary constraints of the method responsible for introducing the action. Indeed, HTN planning systems often model actions as sets of effects and assume that the auxiliary constraints satisfy the unstated preconditions. The end result is that HTN planning provides greater control over what action sequences solve a given problem, but this control comes at the cost of having to write the decomposition methods.

To elaborate, consider a two block *Blocks World* problem initially consisting of a block A on top of block B, and an agent wants to clear a block B using the standard operators unstack(u) or puton(u, v) to respectively move block u onto the Table or block v. Obviously, the action sequence [unstack(A)] solves the problem, but so does any sequence of the form:

$$[\{\texttt{unstack(A),puton(A,B),}\}* \texttt{ unstack(A)}].$$

As far as a PO planner is concerned any of these sequences solves the problem, but only the first is really satisfactory. In this case we could disallow the unsatisfactory solutions by either dropping the puton(u, v) operator or adding preconditions/effects to the operators, but the resulting operator set would not be as intuitively appealing as the standard *Blocks World* operator set.

Although an HTN planner with the computed decomposition methods would suggest the same solutions, the point is that these methods are not computed; a user has explicit control over them. For instance, we already introduced a method for clearing a block, and an HTN planner that uses it would only find the one step solution. In general, our simulation shows that an HTN planner can be complete with respect to a STRIPS problem, but STRIPS is not expressive enough to intuitively represent all of our problems. We often want more control over the action sequences that constitute solutions; thus the STRIPS problem representation is provably less expressive then the HTN representation.

As Erol, Hendler, and Nau [Erol et al. 94a] show, a STRIPS problem can have a solution set with action sequences characterized by a regular expression, but some HTN problems can have a solution set that can only be characterized by a context free grammar. For example, consider a problem that has a solution set with action sequences characterized by $[\texttt{b}^n\texttt{,c,d}^n]$. Since every action b has the same effects, there is no way to make a PO planner avoid suggesting a solution [b,c,d,d], which is not a member of the set. An HTN planner with its abstract tasks and decomposition methods can obviously represent a problem with the appropriate solution set.

2.4.3 Abstraction

Drummond's *hierarchical operator expansion* capability involves the ability to reason about an abstract task and then replace it with a task-network; this comes from an HTN planner's using non-primitive tasks and reduction. To illustrate the utility of reasoning about a task prior to reducing it, consider planning a trip to the supermarket. Commonly, people start by deciding whether to walk or drive, and the next decision is probably the route to take. Notice that these initial decisions are very abstract, and the detail is filled in later. In other words, planning starts at a highly abstract level and later each abstract task becomes more concrete as the lower level details are considered.

As our example illustrates, the use of reduction makes it very natural to try to attribute a level of abstractness with each task. The use of highly abstract tasks enables the avoidance of low level details until reducing the tasks at the appropriate level. The HTN planner O-PLAN[Currie & Tate 91] even facilitates the use of abstraction with the following definition[Tate et al. 94]pg. 1132:

> Each action and effect is introduced at a single DOMAIN MODELING LEVEL and higher level activities introduce activities and effects at the same or a lower level.

Although it is natural to place a task's subtasks at the abstraction level that one less than the task itself, the O-PLAN definition leaves room for the possibility that a subtask can reside at a level that is either equal to or much less than the task's level. The case where a subtask must be at the same level occurs when tasks are recursive; a task can have a copy of itself as a subtask, or a subtask can ultimately reduce to a copy of the task. The alternative case where a subtask is at a much lower abstraction level occurs when a task has multiple subtasks where one can reduce to another but not vice versa; in this case the first is at a higher level than the second, but they both have levels lower than the original task.

Since the difference between two tasks' abstraction levels can depend on an arbitrary number of decomposition methods, assigning abstraction levels to tasks requires globally analyzing all of a problem's decomposition methods. Interestingly, O-PLAN's approach toward assigning a abstraction levels to tasks is similar to an approach toward abstraction has for PO planning. In ALPINE[Knoblock 90] propositions have assigned abstraction levels, and planning at an abstract level is performed by temporarily ignoring goals and preconditions at lower abstraction levels. Once an abstract solution is found, planning drops to a lower level and the plan is extended to handle the goals and preconditions that are no longer ignored. Each proposition's abstraction level depends on the *ordered monotonicity* property, which requires that the truth value of a literal introduced at one abstraction level is unaffected by actions introduced while planning at a lower level. This constraint on propositions' abstraction levels is remarkably similar to that of O-PLAN. Surprisingly, when considering converted STRIPS problems, the two approaches toward abstraction become identical; O-PLAN using its abstraction hierarchy to determine which task to reduce next simulates SNLP using ALPINE to determine which goal to satisfy next.

This correspondence between O-PLAN and SNLP with ALPINE's use of abstraction, lets us take analyses of the latter and apply them to the former. As Smith[Smith & Peot 92] points out, the hierarchies produced by ALPINE can make SNLP perform arbitrarily poorly. The computed abstraction hierarchies only guarantee that plans that cannot be extended to solutions at one level cannot be extended to solutions at lower levels; this guarantee is often called the *downward failure property*. Unfortunately this property does not guarantee a useful abstraction hierarchy; Smith developed one where there are many solutions at one level, but only one can be extended to the next level. Even worse, Yang[Yang 90] illustrates that for some decomposition methods the downward failure property cannot be guaranteed; thus a domain writer must take care when creating a problem's decomposition methods.

2.4.4 Frugal HTN Planning

When given a problem to attain a goal by performing a task, a frugal agent will only perform those sub-actions, of the task, necessary for achieving the goal. For instance, consider a `get-provisions` task to buy a week's worth of groceries from the supermarket. When the only goal is to get a dozen eggs, many of the actions in this generic task become superfluous. The task might include actions to acquire milk and bread, but they do not contribute toward the ultimate goal of having the eggs. A frugal agent would strip out these actions and only buy the eggs.

Frugal HTN planning arose from an attempt to extend HTN planning to deal with ADL style actions. Unlike generic STRIPS, goal attainment problems using ADL actions cannot be directly converted into HTN problems. Where a STRIPS action has a constant set of preconditions, an ADL action's preconditions depends on the effects that must be assured or disabled. For instance, consider an operator for checking out of a supermarket:

$$
\begin{array}{lll}
\texttt{(defoperator} & \texttt{action}: & \texttt{check-out}(u) \\
 & \texttt{precondition}: & \texttt{at}(\texttt{self}, u) \\
 & \texttt{effects}: & \forall v\ \texttt{have}(v) \wedge \texttt{owns}(v, u)\ \texttt{causes}\ \texttt{owns}(v, \texttt{self}) \\
 & & \forall v\ \texttt{have}(v) \wedge \texttt{owns}(v, u)\ \texttt{causes}\ \neg\texttt{owns}(v, u)
\end{array}
$$

This operator has a varying number of preconditions depending on which items the agent wishes to purchase; thus the original approach toward creating a plan fragment with an achieve task for each precondition becomes problematic. The required achieve tasks is not constant.

To resolve the problems caused by ADL actions, we introduce frugal decomposition. Where an HTN task is defined by a method's plan fragment, a *frugal HTN task* is defined to be the part of the plan fragment needed to attain the task's required effects while avoiding undesirable side effects. With this definition, we specify a problem using the maximal plan fragment, and the actual fragment used depends on how the task is used.

2.5 Summary

Chapter 3

FRUGAL HTN PROBLEMS

In this chapter we formalize our intuitive description of frugal hierarchical task-network (FHTN) planning by creating a language to specify problems and interpreting a specification to define its solution plans. In general, our language builds on the developing HTN formalism [Erol et al. 94b, Kambhampati & Hendler 92, Yang 90], but we diverge from this formalism in two related ways. First, we replace the limited STRIPS model of action with a restricted version of Pednault's [Pednault 89, Penberthy & Weld 92] Action Description Language (ADL) to include actions with context sensitive effects. Since these actions' preconditions depend on why they are used, we also diverge from the developing formalism by introducing an ultimate goal for acting.

More precisely, FHTN planning problems are similar to PO and HTN problems in that each has three parts: an initial situation $\mathcal{I}$, a set $\mathcal{O}$ of operators, and a desired behavior description. In both cases the solution is a sequence of actions performable from the initial state. Where a PO problem specifies this behavior with a goal to attain, a HTN problem specifies it using a task to perform. FHTN problems take both tasks and goals to describe behaviors, and solutions are those action sequences that perform the part of a task needed to attain a goal.

Definition 3 *A* FHTN planning problem *is a tuple $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$, which respectively denotes the initial conditions, a set of operators, a set of schemata defining tasks to perform, and a goal to attain.*

Throughout this chapter we develop a FHTN domain from Pednault's briefcase world, which involves transporting objects between locations using a briefcase. For instance, figure 3.1 illustrates a situation in *briefcase world* where all objects are at home, and the paycheck `P` is in the briefcase. Actions in this domain manage the briefcase's contents and moves a briefcase, with its contents, between locations.

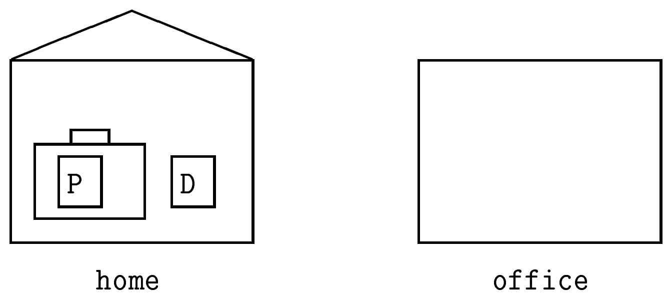


Figure 3.1: The initial situation of our Briefcase World problem

In the first section we define our approach toward representing situations, goals, and actions by discussing Pednault's Action Definition Language (ADL). We then present the developing HTN formalism by comparing task decomposition methods with productions in a graph grammar. In both sections we slightly alter the notation used in order to ultimately define the solutions to FHTN problems in the last section by combining the two formalisms.

3.1 Defining Actions

In general, actions can be characterized as sets of current-situation/next-situation pairs where executing the action in current-situation results in attaining next-situation [Pednault 90]. Using this set, an action is executable in only those situations where a pair specifies the next situation, and more than one pair for the current situation specifies that the action is nondeterministic. Unfortunately this characterization scales poorly in practice since any nontrivial domain can have a potentially infinite number of situations. For this reason a language for describing situations and actions is needed. One important point is that the language used restricts the representable situations and actions.

Prior to ADL, the planning community tended to represent classical planning actions using situation calculus or STRIPS. Although situation calculus enables the representation of very complex actions, its computational intractability prevents its use in all but the simplest planning problems. Alternatively, STRIPS[Fikes & Nilsson 71] can only represent simple actions, but reasoning about an action's effects is much more tractable. We adopt ADL because it provides a tractable middle ground that is more expressive than STRIPS but less intractable than situation calculus.

3.1.1 Situations and Goals

Intuitively, a *situation* is a snapshot, like figure 3.1, of the world at a given moment in time. While illustrations do describe situations, computing plans and proving properties about them requires a more formal representation to encode those features of interest to the reasoning process. Such features involve existing objects, facts, and relationships between objects. Like STRIPS, our situations are sets of positive grounded literals; we represent the one above with the set

$$\{at(\texttt{P}, \texttt{home}), at(\texttt{D}, \texttt{home}), at(\texttt{B}, \texttt{home}), in(\texttt{P})\}.$$

With respect to our problem specification language, we define a *literal* as a string with the form $p(x_1, \ldots, x_k)$ or $\neg p(x_1, \ldots, x_k)$, where p is a predicate, and each term x_i is a constant or variable. In general, we use to symbol ℓ to denote an arbitrary literal, and ℓ is *grounded* if its terms are constant symbols. Just as the grounded literal $at(\texttt{P}, \texttt{home})$ denotes that the paycheck is at home, the predicate of each literal specifies a relationship between its arguments, and all the literals in the situation fully define the relevant relationships. Note that only positive literals appear in the example situation; we make a *closed-world assumption*[Reiter 80] to get the negative literals. Thus, the absence of a positive grounded literal ℓ in a situation implies $\neg\ell$.

To expand our formalism, we define a *goal* as a set of situations where an objective is satisfied. One way to specify a goal is to enumerate the set, but this becomes cumbersome for large sets. We specify goals with formulae from a restricted form of first-order logic and define a goal as the set of situations satisfying its formula; the formula $(at(\mathtt{P}, \mathtt{home}) \wedge at(\mathtt{B}, \mathtt{office}))$ describes our example problem's goal.

Definition 4 *The string ρ is a* FORMULA *when one of the following holds:*

1. *ρ is TRUE or FALSE; or*

2. *ρ is $(x_1 = x_2)$, $(x_1 \neq x_2)$, or $p(x_1, \ldots, x_k)$, with constant or variable x_i; or*

3. *ρ has the form $\neg\rho_1$, $(\rho_1 \wedge \rho_2)$, or $(\rho_1 \vee \rho_2)$, with formulae ρ_1 and ρ_2; or*

4. *ρ has the form $(\exists[v_1, \ldots, v_k; \ell]\rho_1)$ or $(\forall[v_1, \ldots, v_k; \ell]\rho_1)$, with variable v_i, positive literal ℓ containing the variables $v_1, \ldots, v_k$ as terms, and formula ρ_1.*

How we define a literal raises the first restriction on our language; unlike first-order logic, there are no function terms. This facilitates rapid evaluation of literals; determining if a positive or negative ground literal holds in a situation is simply a matter of checking its membership in a set.

Our goal language's second restriction involves each quantified formula's needing a designated literal ℓ to restrict the values that variables can take. While less restrictive than UCPOP's typing each quantified variable, this feature still simplifies reasoning about quantified variables by binding them to constants obtained by matching ℓ against members of a situation. This form of bounded quantification even allows quantification with infinite universes of discourse. For example, the goal specified by $\forall[v; paper(v)]\neg in(v)$ contains those situations with no papers in the briefcase; it is testable when $paper(v)$ is finite.

Formally, a variable v within a formula ρ is *bound* if it only appears as v_i in a sub-formula with the form $(\exists[\ldots v_i \ldots; \ell]\rho_1)$ or $(\forall[\ldots v_i \ldots; \ell]\rho_1)$; otherwise v is *free*. Formulae with no free variables are *closed*, and a closed formula ρ specifies a goal.

Definition 5 *A* GOAL *specified by closed formula ρ is a set of situations $\{s \mid s \models \rho\}$, where the models operator $\models$ is defined by the following eight axioms:*

1. *$s \models TRUE$;*
2. *$s \not\models FALSE$;*
3. *For a ground literal ℓ, $s \models \ell$ iff $\ell \in s$ and $s \not\models \ell$ iff $\ell \notin s$;*
4. *$s \models \neg\rho$ iff $s \not\models \rho$;*
5. *$s \models (\rho_1 \wedge \rho_2)$ iff $s \models \rho_1$ and $s \models \rho_2$;*
6. *$s \models (\rho_1 \vee \rho_2)$ iff $s \models \rho_1$ or $s \models \rho_2$, or both;*
7. *$s \models (\exists[v_1, \ldots, v_k; \ell]\rho_1)$ iff there exists a variable substitution θ such that $\ell\backslash\theta \in s$ and $s \models \rho_1\backslash\theta$; and*
8. *$s \models (\forall[v_1, \ldots, v_k; \ell]\rho_1)$ iff for each variable substitution θ such that $\ell\backslash\theta \in s$, $s \models \rho_1\backslash\theta$.*

For our earlier example goal specification, any situation s where $at(\mathtt{P}, \mathtt{home}) \in s$ and $at(\mathtt{B}, \mathtt{office}) \in s$ is a member of the goal $\{s \mid s \models at(\mathtt{P}, \mathtt{home}) \wedge at(\mathtt{B}, \mathtt{office})\}$ Similarly, those situations where no papers are at the office is specified by the goal $\{s \mid s \models (\forall[v; paper(v)]\neg at(v, \mathtt{office}))\}$.

3.1.2 Actions

To motivate our FHTN formalism, we develop a hierarchical extension of Pednault's *briefcase domain* [Pednault 88], whose operators we encode with three action templates. The `carry` template defines actions to move the briefcase `B` with its contents, while `put-in` and `take-out` respectively define actions to add and remove items from a briefcase (Table 3.1). The conditional and universally quantified effect in `carry` actions denote that the objects moved depend on `B`'s contents (see [Pednault 89, Penberthy & Weld 92] for the action semantics.).

$\texttt{take-out}(u)$	pre :	$in(u)$
	eff :	$\langle TRUE, \neg in(u)\rangle$
$\texttt{put-in}(u, v)$	pre :	$(u \neq \texttt{B}) \wedge at(u, v) \wedge at(\texttt{B}, v) \wedge \neg in(u)$
	eff :	$\langle TRUE, in(u)\rangle$
$\texttt{carry}(v, w)$	pre :	$(v \neq w) \wedge at(\texttt{B}, v)$
	eff :	$\langle TRUE, at(\texttt{B}, w)\rangle, \langle TRUE, \neg at(\texttt{B}, v)\rangle,$
		$\langle (in(u) \wedge (u \neq \texttt{B})), at(u, w)\rangle,$
		$\langle (in(u) \wedge (u \neq \texttt{B})), \neg at(u, v)\rangle$

Table 3.1: Action templates in the briefcase world

As table 3.1 illustrates, ADL actions are similar to those in STRIPS in that they have preconditions and effects; the differences involve how they are represented. Where STRIPS uses a conjunctive set of literals for a precondition, ADL uses an equation from first order predicate logic. Where STRIPS uses a set of positive and negative literals for its effects, ADL can make an effect conditional by tagging it with an extra precondition. In those cases where the extra precondition is not needed, a $TRUE$ is used.

Definition 6 *An* OPERATOR *is a tuple* $\langle f(v_1,\ldots,v_k), \mathsf{pre}_f, \mathsf{eff}_f\rangle$*, where* pre_f *is a formula denoting a precondition, and* eff_f *is a set of effects of the form* $\langle\rho_i,\ell_i\rangle$ *or* $\langle\rho_i,\neg\ell_i\rangle$*. All free variables appearing in precondition formula* pre_f *also appear in* $f(v_1,\ldots,v_k)$*, and each secondary precondition* ρ_i *can have free variables appearing in* $f(v_1,\ldots,v_k)$ *or* ℓ_i*.*

In general, operators are variablized templates defining actions; action $f(c_1,\ldots,c_k)$'s precondition and effects are defined by substituting c_i for v_i in f's operator. For example, the `carry(home,office)` action is defined by respectively substituting `home` and `office` for v and w in `carry`'s operator from table 3.1. After this substitution, the variable u in the last two effects still exists. In general, any variable appearing in ℓ_i, but not in $f(v_1,\ldots,v_k)$, is universally quantified, and $\langle\rho_i,\ell_i\rangle$ denotes a universally quantified effect.

Throughout this dissertation $pre(a)$ denotes an action a's precondition, and $\mathit{eff}(a)$ denotes its effects. In general, $f(c_i,\ldots,c_k)$'s precondition is defined by respectively substituting $c_i,\ldots,c_k$ for $v_1,\ldots,v_k$ in pre_f. Thus we have

$$pre(\texttt{carry(home,office)}) = at(\texttt{B},\texttt{home})$$

implying that the briefcase must be at home before this action executes. As our operator definition implies, an action's precondition is a goal specified by a closed formula.

Like $pre(a)$, computing $\mathit{eff}(a)$ requires taking eff_f and substituting the action's arguments for the operator's parameters, but that is not all – the precondition ρ_i of a universally quantified effect $\langle\rho_i,\ell_i\rangle$ may have free variables. These free variables must be replaced with constants before a universally quantified effect's precondition properly specifies applicable situations. For example, `carry(home,office)` always has the effect $\langle(in(w)\wedge(w\neq\texttt{B})), at(w,\texttt{office})\rangle$ with free variable w.

In general, an action's effects are defined by taking the set eff_f from the appropriate operator and applying the equation

$$\mathit{eff}(a) = \{\langle \rho_i, \ell_i \rangle \backslash \theta \mid \langle \rho_i, \ell_i \rangle \in \mathsf{eff}_f \text{ and } \theta = \{(v_1, c_1) \ldots (v_{ik_i}, c_{ik_i})\}\},$$

where the substitutions (v_j, c_j) come from the action's arguments, v_{ij} is a term of ℓ_i, and c_{ij} is an arbitrary constant. In our example problem there are 5 constants, and the effects of `carry(home,office)` are the set:

$$\{\langle TRUE, at(\mathtt{B}, \mathtt{office})\rangle, \langle TRUE, \neg at(\mathtt{B}, \mathtt{home})\rangle\} \cup$$
$$\{\langle (in(c) \wedge (c \neq \mathtt{B})), at(c, \mathtt{office})\rangle \mid c \in \{\mathtt{P}, \mathtt{B}, \mathtt{home}, \mathtt{office}\}\} \cup$$
$$\{\langle (in(c) \wedge (c \neq \mathtt{B})), \neg at(c, \mathtt{office})\rangle \mid c \in \{\mathtt{P}, \mathtt{B}, \mathtt{home}, \mathtt{office}\}\}.$$

As the goal formulae associated with `carry(home,office)`'s effects illustrate, an ADL action's results can vary with the situation. A *secondary precondition* is an extra requirement that must be satisfied before doing an action in order to control such varying effects. One type of secondary condition, called a *causation precondition*, Σ^a_ℓ, denotes those situations where executing action a adds ℓ to a situation. Since more than one of an action a's effects might add ℓ, causation preconditions are defined by the disjunctive equation

$$\Sigma^a_\ell = \bigvee_{\langle \rho_i, \ell \rangle \in \mathit{eff}(a)} \rho_i. \tag{3.1}$$

For example, suppose we wanted to use a `carry(home,office)` to move the paycheck `P` to the office; Since this action only moves objects in the briefcase, there is a causation precondition that the paycheck is in the briefcase:

$$\Sigma^{\mathtt{carry(home,office)}}_{at(\mathtt{P},\mathtt{office})} = (in(\mathtt{P}) \wedge (\mathtt{P} \neq \mathtt{B})).$$

Unlike *causation preconditions*, a *preservation precondition* Π^a_ℓ denotes situations where executing action a does *not* destructively affect a literal ℓ. In general, this secondary precondition is related to causation preconditions since destructively affecting a literal ℓ is synonymous with adding a literal $\neg\ell$; thus a preservation precondition is defined by the equation

$$\Pi^a_\ell = \neg\Sigma^a_{\neg\ell}. \tag{3.2}$$

In our solution to the briefcase world problem, the preservation precondition for leaving the paycheck at home when moving the briefcase is

$$\Pi^{\texttt{carry(home,office)}}_{at(\texttt{P},\texttt{home})} = \neg(in(\texttt{P}) \wedge (\texttt{P} \neq \texttt{B})).$$

3.1.3 Frugal Action Sequences

Our example operators, together with a description of an initial situation and a goal, constitute a partial order planning problem. For example, suppose that we have an initial situation where a briefcase `B` sits at home and contains a paycheck `P`, see figure 3.1. When we desire to leave the paycheck at home while getting the briefcase to the office, the following action sequence clearly solves our problem:

$$[\texttt{start()},\ \texttt{take-out(P)},\ \texttt{carry(home,office)},\ \texttt{stop()}].$$

In general, we represent an action sequence $[a_0, a_1, \ldots, a_n, a_\infty]$ with the notation $\{a_i\}_{i=0}^n$, where $\{a_i\}_{i=0}^0$ denotes the empty sequence $[a_0, a_\infty]$ with just start and stop actions. While one way to prove that a sequence attains a goal is to compute the result of execution and compare it with the goal, we prove attainment by identifying those conditions that must hold prior to each action and proving that they hold via the *modal truth criterion*. This criterion states that a condition ℓ holds at a point p in the plan if it initially holds or is attained at point q prior to p and persists until p. A causal link $q \xrightarrow{\ell} p$ denotes such a persistence.

With this formalism, showing that $\{a_i\}_{i=0}^n$ attains goal ρ from initial situation $\mathcal{I}$ involves adding actions a_0 with effects from $\mathcal{I}$ and a_∞ with precondition ρ respectively before and after the sequence. Given this augmented action sequence and Pednault's causality theorem [Pednault 86b], proving that $\{a_i\}_{i=0}^n$ attains ρ from $\mathcal{I}$ becomes a matter of finding a set of causal links L such that equation 3.3 holds over all actions in the augmented sequence given that $a_i \prec a_j \prec a_k$.

$$\{\ell \mid i \xrightarrow{\ell} j \in L\} \in \textsc{dnf}\left(pre(a_j) \wedge \left(\bigwedge_{j \xrightarrow{\ell} k \in L} \Sigma_\ell^{a_j}\right) \wedge \left(\bigwedge_{i \xrightarrow{\ell} k \in L} \Pi_\ell^{a_j}\right), \mathcal{I}\right) \tag{3.3}$$

When reasoning about an ADL action's precondition, we defined goals using closed formulae, but the set L only defines a conjunctive set of assured preconditions for any given action. In order to assure an action's ultimate precondition using a conjunctive set, we define the function $\textsc{dnf}(\rho_i, \mathcal{I})$ that takes a closed formula, with the initial situation, and converts it into disjunctive normal form, and returns a disjunctive set of conjunctive sets. For simple formulae involving conjuncts and disjuncts, $\textsc{dnf}(\rho_i, \mathcal{I})$ does not need to refer to $\mathcal{I}$ and has the form:

$$\begin{aligned}
\textsc{dnf}(\rho, \mathcal{I}) = \quad & \{\} && \text{if } \rho = FALSE, \\
& \{\{\}\} && \text{if } \rho = TRUE, \\
& \{\{p(c_1, \ldots, c_k)\}\} && \text{if } \rho = p(c_1, \ldots, c_k), \\
& \textsc{dnf}(\rho_1, \mathcal{I}) \cup \textsc{dnf}(\rho_2, \mathcal{I}) && \text{if } \rho = \rho_1 \vee \rho_2, \\
& \{\phi_1 \cup \phi_2 \mid \ \ \phi_1 \in \textsc{dnf}(\rho_1, \mathcal{I}) \text{ and} && \text{if } \rho = \rho_1 \wedge \rho_2. \\
& \qquad\qquad\quad \phi_2 \in \textsc{dnf}(\rho_2, \mathcal{I})\}
\end{aligned} \tag{3.4}$$

Expanding $\textsc{dnf}(\rho_i, \mathcal{I})$ to handle formulae with negation is simply a matter of pushing the negations down to the literals; this results with the equation:

$$\begin{aligned}
\textsc{dnf}(\rho, \mathcal{I}) = \quad & \{\} && \text{if } \rho = \neg TRUE, \\
& \{\{\}\} && \text{if } \rho = \neg FALSE, \\
& \{\{\neg p(c_1, \ldots, c_k)\}\} && \text{if } \rho = \neg p(c_1, \ldots, c_k), \\
& \textsc{dnf}(\rho_1, \mathcal{I}) && \text{if } \rho = \neg(\neg \rho_1), \\
& \textsc{dnf}((\neg \rho_1) \vee (\neg \rho_2), \mathcal{I}) && \text{if } \rho = \neg(\rho_1 \wedge \rho_2), \\
& \textsc{dnf}((\neg \rho_1) \wedge (\neg \rho_2), \mathcal{I}) && \text{if } \rho = \neg(\rho_1 \vee \rho_2), \\
& \textsc{dnf}((\forall[v_1, \ldots, v_k; \ell]\neg \rho_1), \mathcal{I}) && \text{if } \rho = \neg(\exists[v_1, \ldots, v_k; \ell]\rho_1), \\
& \textsc{dnf}((\exists[v_1, \ldots, v_k; \ell]\neg \rho_1), \mathcal{I}) && \text{if } \rho = \neg(\forall[v_1, \ldots, v_k; \ell]\rho_1).
\end{aligned} \tag{3.5}$$

Finally, extending the function to handle quantification involves inspecting the initial situation $\mathcal{I}$ with the bounding literal ℓ in order to find substitutions for the quantified variables. These substitutions are defined by the equation

$$\Theta(\ell, \mathcal{I}) = \{\theta \mid \ell' \in \mathcal{I} \text{ and } \ell' = \ell\backslash\theta\}, \tag{3.6}$$

and are used to expand the formula over possible substitutions for the quantified variables. For existential and universal quantification the expansion must be disjunctive and conjunctive respectively; thus for quantified formulae $\mathrm{DNF}(\rho_i, \mathcal{I})$ has the form:

$$\begin{aligned} \mathrm{DNF}(\rho, \mathcal{I}) = \;& \mathrm{DNF}(\textstyle\bigvee_{\theta \in \Theta(\ell, \mathcal{I})} \rho_1 \backslash \theta, \mathcal{I}) \quad \text{if } \rho = \exists[v_1, \ldots, v_k; \ell]\rho_1, \\ & \mathrm{DNF}(\textstyle\bigwedge_{\theta \in \Theta(\ell, \mathcal{I})} \rho_1 \backslash \theta, \mathcal{I}) \quad \text{if } \rho = \forall[v_1, \ldots, v_k; \ell]\rho_1, \end{aligned} \tag{3.7}$$

We make one caveat about using a formula's disjunctive normal form to constrain an action's assured preconditions. There is a difference between the conjunctive sets in a formula's DNF and minimal conjunctive sets that would assure the formula. While assuring minimality is NP hard [Garey & Johnson 79], we can still use DNF and L to rule out a lot of suboptimal plans.

Definition 7 *Given the causal links L satisfying equation 3.3 we say that an action sequence $\{a_i\}_{i=0}^n$ is* frugal *if L contains at least one $i \xrightarrow{\ell} j$ for each action a_i.*

In general if their exists an action sequence that solves a problem. Their exists a frugal subsequence solving the problem. Proving this is a matter of taking the links L for existing solution sequence and iteratively removing all actions a_i from $\{a_i\}_{i=0}^n$ without a link $i \xrightarrow{\ell} j$ then all extra links not required by equation 3.3. For instance, the sequence

```
[start(),putin(D), take-out(P), carry(home,office),stop()]
```

solves our example problem, but it has an extraneous step due to no need to move the dictionary `D`.

While frugality rules out action sequences with extraneous actions, there are many other suboptimal action sequences that are still allowed. Consider solving our example problem with the action sequence

```
[start(),take-out(P), putin(P), take-out(P), carry(home,office),stop()].
```

Although the first two actions do not have any real function, frugality cannot rule them out. Since the paycheck must be in the briefcase prior to taking it out, the second action contributes to the ultimate goal.

3.2 Defining Tasks

To gain increased control when defining acceptable action sequences, we turn a PO problem into a FHTN problem by specifying how to attain the goals using tasks and define tasks using a set of methods. Thus FHTN problems take aspects from both STRIPS and HTN planning and specify a BEHAVIOR by denoting that the agent must attain goals by performing a coordinated set of tasks. As each task is defined in terms of subtasks and ultimately denotes a set of finite action sequences, a natural way to characterize them is in terms of a context-free string grammar.

Context-free string grammars define sets of finite character sequences and were originally motivated to describe natural languages. Each grammar contains a distinguished start symbol and three finite sets: variable symbols, terminal symbols, and productions. Each production "$V \rightarrow w$" defines a variable V in terms of a string w of variables and terminals. With this characterization, any variable can be the distinguished start symbol and denote a language consisting of those terminal strings produced by starting with the variable and using productions to replace variables in the string until only terminals are left. For instance, the set of productions $\{S \rightarrow aSb, S \rightarrow ab\}$ makes the variable S denote the language $a^n b^n$ where both a and b are terminal symbols.

3.2.1 Task Network Grammars

As shown in chapter 2 decomposition methods define tasks in terms of coordinated sets of subtasks instead of sequences of subtasks. Thus methods are much like productions in the flow grammars introduced by Brotsky [Brotsky 84] as generalizations of context-free string grammars.

In general $\tau_{\mathtt{s}}$ denotes a problem task, and "$\tau \Rightarrow$ *network*" denotes a method with *network* specifying the coordinated subtasks of task τ. Boxes within a network represent tasks, while literals above a task denote its possible preconditions. Each precondition has one arrow pointing to it, and each arrow denotes how to attain a precondition. Thus figure 3.2 requires than any solution to the problem moves the briefcase with a `move-to(office)` task, and the paycheck stays at home between the `do[start()]` and `do[stop()]` primitive tasks. These primitives always appear in decomposition methods for problem tasks; they respectively denote the actions a_0 and a_∞ before and after actual action sequence.

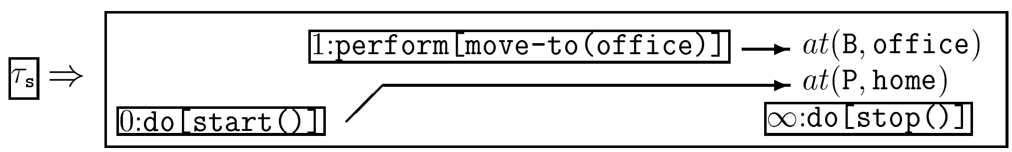


Figure 3.2: The decomposition for the start task of our example problem

In addition to $\tau_{\mathtt{s}}$ there are three classes of tasks: primitive tasks denoting actions to execute, compound tasks denoting procedures to perform, and goal tasks denoting literals to attain. The strings (n:`do[`$f(c_1, \ldots, c_k)$`]`) and (n:`perform[`$t(c_1, \ldots, c_k)$`]`) respectively denote a primitive task to execute action $f(c_1, \ldots, c_k)$, and a compound task to perform $t(c_1, \ldots, c_k)$. Each term c_i is a constant, and the symbols f and t respectively denote the names of operators and compound tasks. Goal tasks are similarly denoted by strings of the form (n:`achieve[`ℓ`]`), where ℓ is a literal of the form $p(c_1, \ldots, c_k)$ or $\neg p(c_1, \ldots, c_k)$ and p is a predicate.

In terms of our context-free grammar primitive tasks denote terminals and each *nonprimitive* task is a variable corresponding to one of a collection of coordinated sets of subtasks. in our example problem, the `perform[move-to(office)]` task ultimately corresponds to two primitive tasks to manage the contents of the briefcase and carry it to the office.

3.2.2 Perform Tasks

As we develop our FHTN formalism, we will define (1:`perform[move-to(office)]`) to get a set of objects to the office; we also define procedures `load(`u`)` and `unload(`u`)` to control that set. Where `load(`u`)` loads the briefcase with an object u, `unload(`u`)` removes it. As figure 3.3 shows, `unload(`u`)` uses a subtask to take u out of the briefcase; the $in(u)$ appearing above the subtask requires that u is in the briefcase prior to taking it out. Since the arrow leading into $in(u)$ starts at a point protection[1], attainment is assumed; thus this method only applies to cases where u is currently in the briefcase.

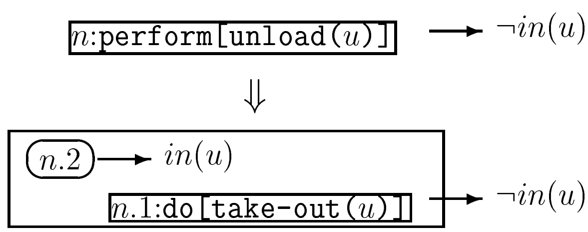


Figure 3.3: An abstract task for unloading an object

As shown in figure 3.3, n is a dotted sequence of symbols denoting a unique label for a task or point protection; this label has two components, a prefix relating it to it's parent task and a suffix differentiating it from labels of sibling subtasks. In general, we take advantage of a label's uniqueness by unambiguously referring to a task $\tau_n = (n : \alpha)$ by its label n; we use the symbols τ_n and ℓ to represent task strings and literal strings respectively.

More precisely, a non-primitive task τ_n's coordinated set of subtasks is specified using a contextualized task network, and a *contextualized task network* (CTN) is a

[1] Symbolically, a point protection looks like a causal link $n \xrightarrow{p} m$, but τ_n is unspecified.

tuple $\langle P : T, B, O, L : E\rangle$, where $\langle T, B, O, L\rangle$ is a task network, P is a set of constructs $(\xrightarrow{\ell} n.i)$ denoting how subtasks use a task's preconditions, and E is a set of constructs $(n.i \xrightarrow{\ell})$ denoting how subtasks produce a task's effects. The label $n.i$ either denotes a subtask or a point protection depending respectively on whether or not $\tau_{n.i} \in T$. Finally, decomposition methods specify a task's coordinated set of subtasks.

Definition 8 *A* DECOMPOSITION METHOD *for a non-primitive task τ is a construct $[\tau \Rightarrow ctn]$, where ctn is a CTN specifying τ's coordinated set of subtasks and how they fulfill τ's contextual requirements.*

In an HTN problem, variablized schemata from the set $\mathcal{M}$ define decomposition methods for compound tasks of the form $(n{:}\texttt{perform[}t(c_1, \ldots, c_k)\texttt{]})$; for instance, we define the decomposition method for $(1.4.\texttt{P}{:}\texttt{perform[unload(P)]})$ by replacing n and u with $1.4.\texttt{P}$ and $\texttt{P}$ respectively in the schema:

$$
\begin{aligned}
(n{:}\texttt{perform[unload(}u\texttt{)]}) \Rightarrow \langle\ & P_n = \{\} : \\
& T_n = \{(n.1{:}\texttt{do[take-out(}u\texttt{)]})\}, \\
& B_n = \{\}, \\
& O_n = \{\}, \\
& L_n = \{(n.2 \xrightarrow{in(u)} n.1)\} : \\
& E_n = \{(n.1 \xrightarrow{\neg in(u)})\}\ \rangle.
\end{aligned}
$$

Just as operators in $\mathcal{O}$ have variablized names like $f(v_1, \ldots, v_k)$ to define actions like $f(c_1, \ldots, c_k)$, methods in $\mathcal{M}$ specify the coordinated subtasks for variablized tasks like $(n{:}\texttt{perform[}t(v_1, \ldots, v_k)\texttt{]})$ and can define the subtasks for any task.

In the rest of this section, we define $\mathcal{G}(\mathcal{M})$ as the set of decomposition methods for tasks like $(n{:}\texttt{perform[}t(c_1, \ldots, c_k)\texttt{]})$, $\mathcal{G}(\mathcal{M})$ has a set of *perform methods* obtained by taking the appropriate schemata of $\mathcal{M}$ and replacing variables with constants from $\mathcal{C}$, where $\mathcal{C}$ contains all constants referred to by the problem. In order to formally define $\mathcal{G}(\mathcal{M})$, we use the equation

$$\mathcal{G}(\mathcal{M}) = \{[\tau \Rightarrow ctn]\backslash\{(v_1, c_1), \ldots, (v_k, c_k)\} \mid [\tau \Rightarrow ctn] \in \mathcal{M} \text{ and } c_i \in \mathcal{C}\},$$

where $\{v_1, \ldots, v_k\}$ are the free variables in method $[\tau \Rightarrow ctn]$.

Having one variable, (n:`perform[unload(`u`)]`)'s schema corresponds to $|\mathcal{C}|$ methods; in general, a schema with n variables corresponds to $|\mathcal{C}|^n$ methods. Since a task's name might not mention some of a schema's variables, a single schema in $\mathcal{M}$ can denote multiple methods to decompose a task. For instance, our schema defining (n:`perform[load(`u`)]`) denotes $|\mathcal{C}|$ methods for (1:`perform[load(P)]`); the appropriate one to use depends on the paycheck's location.

Similar to `unload(`u`)`, `load(`u`)` (figure 3.4) only applies when u is outside the briefcase, but this new schema uses two extra features of our task description formalism: constraints and achieve tasks. In general, extra annotations like the constraint "$u \neq$ `B`" appear in the oval; here the constraint assures that `load(B)` never happens. This constraint also assures that the only way to attain $at(\mathtt{B}, v)$, and that is with an achieve task. Finally, we do not explicitly define `achieve[]` tasks; they are defined in terms of the other `perform[]` tasks.

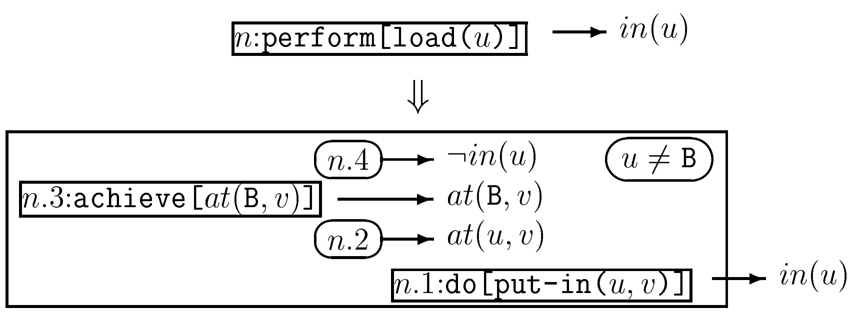


Figure 3.4: An abstract task for loading an object

3.2.3 Achieve Tasks

Although we only specify `perform[`$t(v_1, \ldots, v_k)$`]` schemata in $\mathcal{M}$, we often refer to `achieve[`ℓ`]` tasks. In general, the definitions for these tasks are implicit; currently the planning community is investigating two competing approaches toward defining an `achieve[`ℓ`]` task. One defines it as any action sequence attaining ℓ[Young et al. 94], and the other defines it as either an empty *phantom* action sequence or any `perform[`$t(c_1, \ldots, c_k)$`]` task with ℓ as an effect[Tate et al. 94]. Although we take the latter approach, both can be characterized in terms of computing a set of methods $\mathcal{G}(\mathcal{A}(\mathcal{M}))$, where we define $\mathcal{A}(\mathcal{M})$ with the union

$$\{[(n{:}\texttt{achieve[}\ell\texttt{]}) \Rightarrow \langle \emptyset : \emptyset, \emptyset, \emptyset, \emptyset : \{(n.1 \overset{\ell}{\rightarrow})\}\rangle]\} \cup$$
$$\{[(n{:}\texttt{achieve[}\ell\texttt{]}) \Rightarrow \langle \emptyset : tn_n : \{(n.i \overset{\ell}{\rightarrow})\}\}\rangle \mid \; [\tau_n \Rightarrow \langle P_n : tn_n : E_n\rangle] \in \mathcal{M} \text{ and } (n.i \overset{\ell}{\rightarrow}) \in E_n\}.$$

For example, figure 3.5 illustrates two achieve schemata for (n:`achieve[`$\neg in(o)$`]`) tasks: figure 3.5a phantomizes the task and figure 3.5b expands it using the previously described method for `perform[unload(`o`)]`. Notice that our approach toward phantomization does not need null `do[no-op]` tasks like previous HTN planning formalisms; this is one of the positive ramifications of our approach toward representing point protections. Finally, since all non-primitive tasks are either achieve or perform tasks, all nonprimitive tasks are decomposed using schemata from $\mathcal{M} \cup \mathcal{A}(\mathcal{M})$.

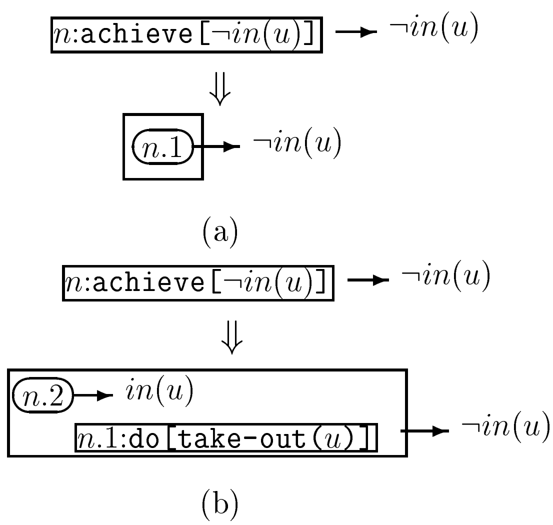


Figure 3.5: Two methods from $\mathcal{A}(\mathcal{M})$ for removing u from the briefcase.

3.2.4 Universal Variables

We build on the developing HTN formalism in two related ways: conditional expansion and universal quantification. Until now HTN planners used STRIPS to model the preconditions and effects of both actions and tasks. They assumed that actions have constant sets of preconditions and effects and decomposition methods specified constant sets of subtasks. By using ADL to model actions and tasks, we remove this assumption. Not only do an action's preconditions and effects depend on how it is used, but so does a decomposition method's set of subtasks. Our decomposition methods specify the maximal set of subtasks, but a task is defined in terms of those subtasks needed to attain desired effects while avoiding undesirable side effects.

Since an action can have universally quantified effects and preconditions, we have to introduce universal quantification to decomposition methods. For instance, consider a task (n:`perform[move-to(`w`)]`) for moving a set of objects to a location w like the office. As shown in figure 3.6, it consists of a primitive task to carry the briefcase to the office preceded by a set of achieve tasks that appropriately manage the contents of the briefcase. We introduce universally quantified variables, like $\mathbf{u}$ in the figure, as a shorthand to specify the maximal set of subtasks in cases like managing the briefcase's contents. Throughout this dissertation, universal variables will appear in boldface type.

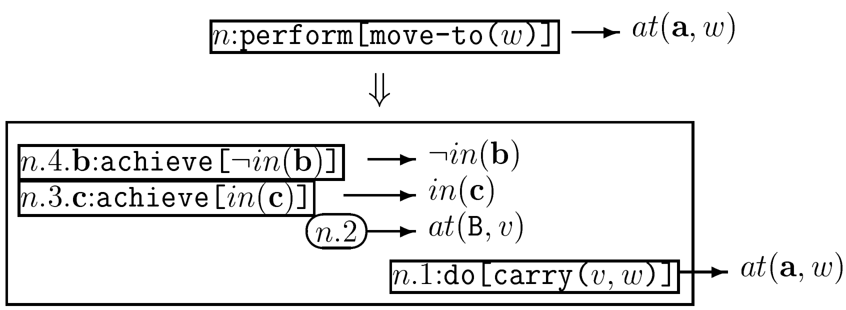


Figure 3.6: An abstract task for moving objects to a location w

Essentially, a universal variable in a method denotes that each element of any set in the method's CTN expands to $|\mathcal{C}|$ elements. With respect to **a**, **b**, and **c** in figure 3.6, the CTN is:

$$
\begin{aligned}
\langle P_n &= \textstyle\bigcup_{\mathbf{a},\mathbf{b},\mathbf{c}\in\mathcal{C}}\{\} : \\
T_n &= \textstyle\bigcup_{\mathbf{a},\mathbf{b},\mathbf{c}\in\mathcal{C}}\{(n.1{:}\texttt{do[carry(}v,w\texttt{)]}), \\
&\qquad\qquad (n.3.\mathbf{c}{:}\texttt{achieve[}in(\mathbf{c})\texttt{]}), \\
&\qquad\qquad (n.4.\mathbf{b}{:}\texttt{achieve[}\neg in(\mathbf{b})\texttt{]})\}, \\
B_n &= \textstyle\bigcup_{\mathbf{a},\mathbf{b},\mathbf{c}\in\mathcal{C}}\{\}, \\
O_n &= \textstyle\bigcup_{\mathbf{a},\mathbf{b},\mathbf{c}\in\mathcal{C}}\{(n.0 \prec n.1), (n.1 \prec n.\infty), \\
&\qquad\qquad (n.0 \prec n.3.\mathbf{c}.0), (n.3.\mathbf{c}.0 \prec n.3.\mathbf{c}.\infty), (n.3.\mathbf{c}.\infty \prec n.\infty), \\
&\qquad\qquad (n.0 \prec n.4.\mathbf{b}.0), (n.4.\mathbf{b}.0 \prec n.4.\mathbf{b}.\infty), (n.4.\mathbf{b}.\infty \prec n.\infty)\}, \\
L_n &= \textstyle\bigcup_{\mathbf{a},\mathbf{b},\mathbf{c}\in\mathcal{C}}\{(n.4.\mathbf{b} \overset{\neg in(\mathbf{b})}{\rightarrow} n.1), (n.3.\mathbf{c} \overset{in(\mathbf{c})}{\rightarrow} n.1), (n.2 \overset{\texttt{at(B,}v\texttt{)}}{\rightarrow} n.1)\} : \\
E_n &= \textstyle\bigcup_{\mathbf{a},\mathbf{b},\mathbf{c}\in\mathcal{C}}\{(n.1 \overset{at(\mathbf{a},w)}{\rightarrow})\}\rangle.
\end{aligned}
$$

Notice how each set is unioned over $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathcal{C}^3$, and each element only has one universal variable; thus each union is over $|\mathcal{C}|$ sets, and each item with **a**, **b**, or **c** expands to $|\mathcal{C}|$ items.

Formally, we define $\mathcal{U}(\mathcal{M})$ as the set of methods computed by expanding the universal variables mentioned in $\mathcal{M}$. Each set S is expanded into the set

$$\{e\backslash b \mid e \in S \text{ and } b \text{ is a grounding substitution for universal variables in } e\}.$$

Looking back at our example, the two regular variables v and w implies that it defines $|\mathcal{C}|^2$ perform methods in $\mathcal{G}(\mathcal{U}(\mathcal{M}))$, and each method can have $O(|\mathcal{C}|)$ elements in its component sets.

Given that universally quantified variables appropriately expand decomposition methods to define tasks and handle actions with universally quantified preconditions and effects, the resulting expansion contains many subtasks that are not needed to solve the problem. Even worse, the full expansion of our example is inconsistent due to the requirement that $in(\mathtt{P})$ and $\neg in(\mathtt{P})$ holds simultaneously prior to carrying the briefcase.

In order to avoid this problem, we define a task in terms of those subtasks needed to attain its effects while avoiding unwanted side effects. Thus the methods in $\mathcal{M}$ now define a maximal CTN ; the actual network might have fewer tasks and auxiliary constraints depending on what is required of the original task. To facilitate this, we define $\mathcal{G}^*(\mathcal{M})$ to be a set of partial ground methods, and

$$\begin{aligned}\mathcal{G}^*(\mathcal{M}) = \{\ [\tau_n \Rightarrow \langle P' : T', B_n, O_n, L' : E' \rangle] \mid & \\ & [\tau_n \Rightarrow \langle P_n : T_n, B_n, O_n, L_n : E_n \rangle] \in \mathcal{G}(\mathcal{U}(\mathcal{M} \cup \mathcal{A}(\mathcal{M}))), \\ & P' \subseteq P_n, T' \subseteq T_n, L' \subseteq L_n, \text{ and } E' \subseteq E_n\}.\end{aligned}$$

For example, if we use task $(n{:}\mathtt{perform[move\text{-}to(office)]})$ to get a dictionary D to the office, the subtask $(n.3.\mathtt{D}{:}\mathtt{achieve[}in(\mathtt{D})\mathtt{]})$ is needed to attain the desired effect, but no other subtask $\tau_{n.3.o}$ is needed to put another object o into the briefcase. Similarly, since we wish to keep the paycheck at home $(n.2.\mathtt{P}{:}\mathtt{achieve[}\neg in(\mathtt{P})\mathtt{]})$ is needed to avoid an undesired effect, but no other subtask $\tau_{n.2.o}$ is needed to get another object o out of the briefcase.

The last step in producing methods from the schemata involves removing the label prefix variable n from the schemata. To do this we define the function $\mathcal{N}(\mathcal{M})$ with the purpose to replace the variable label n in a method $[(n{:}\alpha) \Rightarrow ctn_n]$ with an actual label – like "1.4.P". Obviously, since the number of labels is countably infinite, so is the cardinality of methods in $\mathcal{N}(\mathcal{M})$. The only methods that $\mathcal{N}$ does not expand to a countably infinite set are the ones for $\tau_{\mathtt{s}}$. Finally we now define $G(\mathcal{M})$ to denote an FHTN problem's set of methods and

$$G(\mathcal{M}) = \mathcal{N}(\mathcal{G}^*(\mathcal{M})), \tag{3.8}$$

3.2.5 Linguistic Restrictions

Since both preconditions and supporting tasks can have universal variables, they both represent sets; the question is, "which task supports which precondition?" Our example's auxiliary constraints points toward a natural answer; when the same variable appears in both the precondition and its supporting task, that variable constrains which actual supporting task to use for a particular precondition. In order to uniquely determine the supporting task, we introduce a syntactic limitation on our representation language; in $n \xrightarrow{\ell} m$ or $n \xrightarrow{\ell}$ the universal variables appearing in n must also appear in ℓ. In our example, both n and ℓ had the same universal variable $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$. Given universally quantified effects, a task can support a whole set of preconditions; thus universal variables in ℓ need not appear in n.

Given our characterization of universal variables, there are also restrictions on where they can appear in binding and ordering constraints. They must never appear in a binding constraint; otherwise the constraints in B_n are inconsistent. With respect to ordering constraints, the universal variables appearing on one side of a constraint must be a subset of those appearing on the other side; while this restriction is not required for consistency, it simplifies our algorithms and does not restrict representable orderings. If m and n are two labels containing overlapping universal variables U_n and U_m, $\{(m \prec n)\}$ is equivalent to $\{(m \prec o), (o \prec n)\}$ where o has the universal variables $U_n \cap U_m$.

Another ordering constraint restriction involves nonprimitive tasks. If τ_n is a nonprimitive subtask of $\tau_{a(n)}$, then $O_{a(n)}$ cannot have ordering constraints referring directly to τ_n; they must either refer to the moment $n.0$ immediately prior to τ_n or $n.\infty$ immediately after τ_n. Also every set of ordering constraints O_n must order all tasks in T_n between the moments $n.0$ and $n.\infty$. Point protections do not have to be ordered with respect to any label.

Our final linguistic restriction regards the use of existential variables in a task's preconditions and effects. We require that any variable used existentially in a precondition or effect also appear as a parameter of the task. For instance, a blocksworld action to move a block onto another must be encoded as `puton`($block, from, to$). This differs from the standard approach; $from$ must be a parameter. We make this requirement in order to properly instantiate a task or action into the context of an HTN.

3.3 Solutions

Taking aspects from both STRIPS and HTN planning, an FHTN planning problem specifies a BEHAVIOR by denoting that the agent must attain a goal ρ by performing $\tau_{\mathtt{s}}$'s coordinated set of subtasks according to their definitions in $\mathcal{M}$. As in HTN planning, $\mathcal{M}$ tells an agent what to do; as in STRIPS planning, the goal ρ tells an agent why to act. To conform with a behavior, an agent executes a sequence of actions that performs that part of the coordinated tasks which contributes toward the goal of acting.

3.3.1 From Grammars to Languages

In any form of HTN planning, a task-network grammar denotes a set of rewrite rules, where each rule describes how to replace a non-primitive task with a subtask network. Just as a string grammar gradually rewrites a single non-terminal symbol into a progressively longer string until all symbols are terminals, a task-network grammar gradually rewrites a single task as a progressively larger network until all tasks are primitive. While a string grammar defines a set of terminal symbol strings, a task-network grammar defines a set of primitive task networks, where a *primitive task network* is a task network consisting solely of primitive tasks. For instance, figure 3.7 illustrates the primitive task network for our example problem's behavior.

Just as a string grammar associates sets of terminal strings with each non-terminal

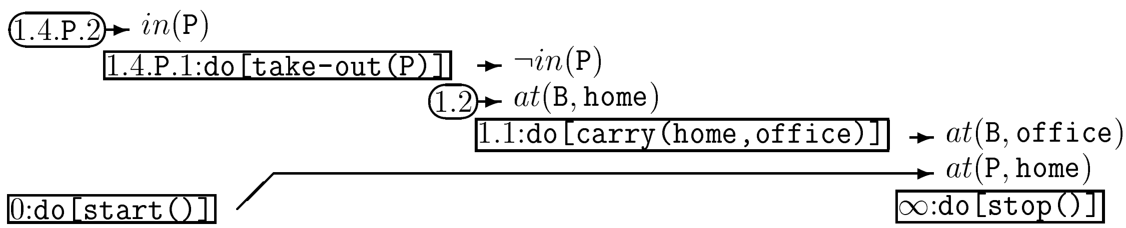


Figure 3.7: A primitive task network solving our example problem

symbol, a task-network grammar associates a task with a set of related primitive CTNs . We now formally define the set by developing a notation to represent a derivation. This notation requires two relations $\Rightarrow$ and $\stackrel{*}{\Rightarrow}$ defined from the set of methods in $G(\mathcal{M})$. If $[\tau_n \Rightarrow \langle P_n : T_n, B_n, O_n, L_n : E_n\rangle]$ is a decomposition method in $G(\mathcal{M})$, then

$$\langle P : T, B, O, L : E\rangle \Rightarrow \langle P' : (T - \{\tau_n\}) \cup T_n, B \cup B_n, O' \cup O_n, L' \cup L_n : E'\rangle,$$

where the primed sets are computed by redirecting constraints involving τ_n toward its subtasks. These sets are defined by the four equations:

$$\begin{aligned}
P' &= \{(\xrightarrow{Q} n.i) \mid (\xrightarrow{Q} n) \in P \text{ and } (\xrightarrow{Q} n.i) \in P_n\} \cup \\
&\quad\ \{(\xrightarrow{Q} m) \mid (\xrightarrow{Q} m) \in P \text{ and } m \neq n\} \\
O' &= \{(m_1 \prec m_2) \mid (m_1 \prec m_2) \in O \text{ and } m_i \neq n\} \cup \\
&\quad\ \{(n.i \prec m) \mid (n \prec m) \in O \text{ and } \tau_{n.i} \in T_n\} \cup \\
&\quad\ \{(m \prec n.i) \mid (m \prec n) \in O \text{ and } \tau_{n.i} \in T_n\} \\
L' &= \{(m_1 \xrightarrow{Q} m_2) \mid (m_1 \xrightarrow{Q} m_2) \in L \text{ and } m_i \neq n\} \cup \\
&\quad\ \{(n.i \xrightarrow{Q} m) \mid (n \xrightarrow{Q} m) \in L \text{ and } (n.i \xrightarrow{Q}) \in E_n\} \cup \\
&\quad\ \{(m \xrightarrow{Q} n.i) \mid (m \xrightarrow{Q} n) \in L \text{ and } (n.i \xrightarrow{Q}) \in P_n\} \\
E' &= \{(n.i \xrightarrow{Q}) \mid (n \xrightarrow{Q}) \in E \text{ and } (n.i \xrightarrow{Q}) \in E_n\} \cup \\
&\quad\ \{(n \xrightarrow{Q}) \mid (m \xrightarrow{Q}) \in E \text{ and } m \neq n\}.
\end{aligned}$$

We say that the decomposition method is applied to the first CTN to obtain the second or that the first *directly derives* the second with the set $G(\mathcal{M})$. Two CTNs are related by $\Rightarrow$ iff the second is obtained from the first by one application of a method in $G(\mathcal{M})$.

Given a sequence of $m \geq 1$ contextualized task networks [ctn_1,...,ctn_m] such that $ctn_i \Rightarrow ctn_{i+1}$ for $G(\mathcal{M})$ and each i between 1 and m, we say that $ctn_1 \stackrel{*}{\Rightarrow} ctn_m$ or ctn_1 *derives* ctn_m. Thus $\stackrel{*}{\Rightarrow}$ is the reflexive and transitive closure of $\Rightarrow$. Expanding $\Rightarrow$ and $\stackrel{*}{\Rightarrow}$ into relations from tasks or CTNs to CTNs is a simple matter of including $\tau \Rightarrow ctn$ if and only if $[\tau \Rightarrow ctn] \in G(\mathcal{M})$. Occasionally we will write $ctn \stackrel{d}{\Rightarrow} ctn'$ to denote that ctn directly derives ctn' using decomposition method $d \in G(\mathcal{M})$.

The language $\mathcal{L}(\mathcal{M})$ generated by $\mathcal{M}$ is the set

$$\{ctn \mid \; ctn \text{ is primitive and } \tau_{\mathsf{s}} \stackrel{*}{\Rightarrow} ctn\}.$$

That is, a task network is in $\mathcal{L}(\mathcal{M})$ if it is derivable from τ_{s} using methods in $G(\mathcal{M})$ and all its tasks are primitive.

3.3.2 Solution Sequences

In HTN planning, we perform a coordinated set of primitive tasks by executing each task's action while maintaining the coordination constraints. Thus we perform these tasks by ordering their actions into a sequence that we can execute from the initial situation while maintaining the coordination constraints. For instance, the primitive CTN solving our example problem appeared in figure 3.7; ordering these primitive tasks results in the action sequence

`[start(), take-out(P), carry(home,office), stop()]`.

In general, the first and last actions of a solution sequence are respectively the null `start()` action and the null `stop()` action. As such, a FHTN is performed by an action sequence `[`$a_0, a_1, \ldots, a_m, a_\infty$`]`, where a_0 is `start()` and a_∞ is `stop()`. Just as in our discussion of frugal action sequences, we use these null tasks to facilitate satisfying auxiliary conditions involving the problem's initial state and goal. For instance, our example problem required maintaining the location of the paycheck between beginning and end of the action sequence.

To be precise, a sequence performs a primitive CTN $\langle \emptyset : tn : \emptyset \rangle \in \mathcal{L}(\mathcal{M})$ when two conditions hold (1) there exists a matching π between the sequence's actions and the primitive tasks in tn such that $\pi(n) = i$ when $\tau_n = (n{:}\texttt{do[}a_i\texttt{]})$ and (2) executing the sequence does not violate the coordinating constraints. Essentially, our using $\mathcal{L}(\mathcal{M})$ formalizes the progressive decomposition of non-primitive tasks in order to derive a solution primitive task network, and our using π formalizes our ordering the primitive task's actions into a sequence.

Since tn is a task network, the coordinating constraints fall into three sets: binding constraints B, ordering constraints O, and Auxiliary constraints L. By far the binding constraints are easiest to assure. Since the methods in $G(\mathcal{M})$ lack variables, so does tn; thus the binding constraints have the form $(c_i = c_j)$ or $(c_i \neq c_j)$ and are respectively satisfied when c_i and c_j denote the same symbol or different symbols. Unlike the binding constrains, the ordering constraints qualify the matching π; each ordering constraint $(n \prec m)$ is satisfied iff $\pi(n) < \pi(m)$.

Definition 9 *Action sequence $\{a_i\}_{i=0}^{k}$ solves FHTN planning problem $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$ when:*

1. *it attains ρ when executed from $\mathcal{I}$ with a set of causal links L' satisfying equation 3.3,*

2. *it performs a task network $\langle \emptyset : T, B, O, L : \emptyset \rangle$ found in $\mathcal{L}(\mathcal{M})$ with a 1-1 relation $(i \xrightarrow{\ell} j, n \xrightarrow{\ell} m)$ between elements of L' and L such that $j = \pi(m)$ always holds and $i = \pi(n)$ when $\tau_n \in T$, and*

3. *there exists $n \xrightarrow{\ell} m \in L$ for each $\tau_n \in T$ other than τ_0 and τ_∞.*

Intuitively, these three conditions correspond to: (1) the sequence is frugal and attains the goal when executed from the initial state, (2) the sequence only performs actions from the problem's tasks, and (3) each action has an explicit reason for being included in its task. This explicit reason implies that an action sequence is frugal, but differs slightly due to the semantics underlying point protections. Since a point protection should be satisfied by an existing task, adding tasks to satisfy point protections is illegal. The third condition enforces this by requiring the existence of an auxiliary constraint $n \xrightarrow{\ell} m$ denoting a legal reason for including τ_n.

3.4 Summary

This chapter has formally defined a frugal hierarchical task-network problems and what it means for action sequences to solve them. We begin by reviewing the ADL action representation as implemented by UCPOP, and then define frugal action sequences that attain a given goal.

In our definition, a frugal action sequence not only attains a goal, but each of its actions contributes something to the attainment. These contributions are characterized in a sequence's causal structure, denoted by a set of causal links. In this set, actions and the initial conditions support links, and links support goals and actions' required preconditions. An action's contribution is denoted by those links that it supports, and its required preconditions assure that the action provides its contribution while not clobbering other actions' contributions.

To gain increased control over which action sequences denote acceptable problem solutions, we present a framework for specifying languages to define acceptable action sequences. This framework extends on the emerging hierarchical task network (HTN) formalism. In this formalism, the problem is to perform some task given the initial conditions. The formalism uses a grammar to denote action sequences that perform the problem's task, and the search is to find a sequence executable from the initial conditions.

Instead of attaining a goal by any means or blindly performing a task, the solution to a frugal hierarchical task network (FHTN) problem performs a task adapted to attain a goal. Our extensions are driven by an ADL action's universally-quantified conditional effects and its required precondition's depending on the context of its use. Our extended framework can define tasks in terms of universally quantified sets of subtasks which are conditionally included when required. Unlike HTN planning, a task's subtasks depend on both its desired effect and a wish to avoid clobbering the work of other tasks.

Chapter 4

THE UCPARSE ALGORITHM

This chapter describes the UCPARSE algorithm for solving FHTN planning problems. As previously mentioned, a solution plan satisfies two constraints; one involves performing the problem's tasks as specified by $\mathcal{M}$, and the other assures that each step contributes toward attaining the problem's goals as specified by ρ. Either constraint can drive plan synthesis while the other limits it. In the following 3 sections, we show: how goals can drive the planning process; how UCPARSE uses goals and frugality to drive plan generation; and how derivation maintenance uses tasks to limit plan generation.

4.1 Precondition Achievement

Since UCPARSE is a descendent of UCPOP[Barrett et al. 93], there are more than a few similarities between the two algorithms; they both use precondition achievement to synthesize plans while computing an action's preconditions to assure and disable effects. Used by most partial order planners, precondition achievement involves taking a partially synthesized plan, selecting an unsatisfied precondition, nondeterministically choosing how to satisfy it, and repeating until no unsatisfied preconditions remain. For instance, consider our briefcase world problem to move the briefcase to the office; the flaw with a zero step plan is that it leaves the briefcase at home. A possible way to satisfy this precondition is to add a `carry` task to the plan for moving the briefcase.

More precisely, each precondition p of primitive τ_n is an *open condition* if the plan has no corresponding causal link $m \xrightarrow{p} n$ denoting how the precondition is satisfied. During precondition achievement, an open condition $\langle p@n \rangle$ is selected and resolved by adding a causal link $m \xrightarrow{p} n$ to the plan. The algorithm might also add task τ_m if it did not already exist in the plan; In our example, we added a `carry` task.

When adding a task or link to the plan, it may interact with other parts of the plan; we say that τ_o *threatens* a causal link $m \xrightarrow{p} n$ if it can cause $\neg p$ while occurring between τ_m and τ_n. UCPARSE follows UCPOP's example and resolves threats by modifying the plan to assure that either τ_o cannot cause $\neg p$ or cannot occur between τ_m and τ_n.

To start the precondition achievement process, a problem is encoded as an *null plan* $\langle T, B, O, L \rangle$, where T has two tasks (0:`do[start()]`) and (∞:`do[stop()]`). The `start()` action has the initial conditions for its effects, and `stop()` has the problem's goals for preconditions. There are no auxiliary constraints in L, but B and O come directly from the method defining $\tau_{\mathtt{s}}$ in $\mathcal{M}$. One such constraint in O forces τ_0 before τ_∞. The set of operators $\mathcal{O}$ defines those primitive tasks that we can add to T to satisfy open conditions, and the objective is to search through the plans synthesized by satisfying open conditions until a plan with no open conditions is found. For example, we show the null plan and its first two successors in figure 4.1. In all cases, a plan's open conditions appears below it.

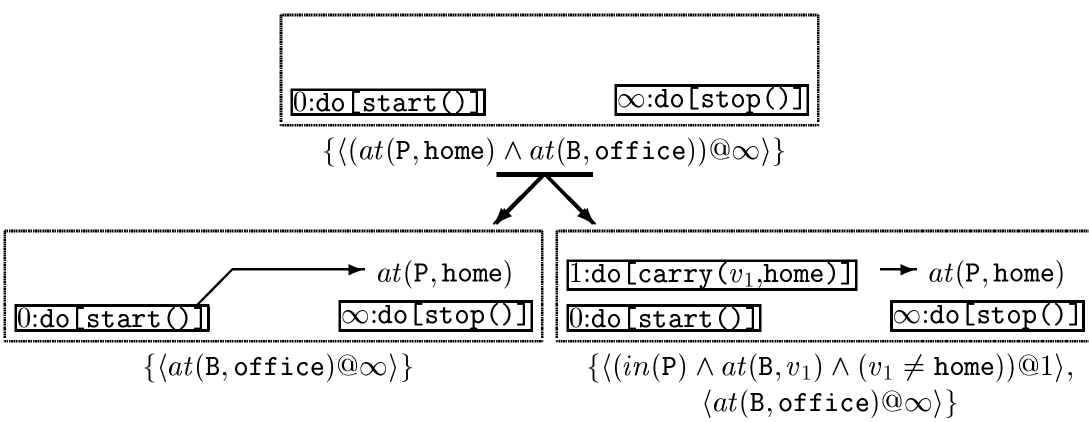


Figure 4.1: First three nodes of the precondition achievement search space.

While one open condition is satisfied at each point during the precondition achievement process, the repair might result in a new set of preconditions. With respect to figure 4.1, the added `carry` step has a precondition requiring that the briefcase is at some location v_1. Eventually, all preconditions get satisfied, and the resultant plan solves the problem. The point where UCPARSE differs from UCPOP involves using tasks and decomposition constraints to restrict how preconditions are satisfied.

As alluded to in the previous chapter, a problem's coordinated set of tasks always includes unique `do[start()]` and `do[stop()]` tasks; they correspond with the `start()` and `stop()` steps of the null plan. By enforcing a correspondence between links added during precondition achievement and auxiliary constraints in the problem's coordinated set of tasks, UCPARSE restricts how particular open conditions are satisfied. For instance, figure 4.1 illustrates that precondition achievement suggests two possible ways to get the paycheck home. One adds a `carry` step to move the paycheck, and the other adds a link to τ_0 assuring that the paycheck stays home. While UCPOP explores both alternatives, UCPARSE never generates the first because it does not match the FHTN planning problem's coordinated tasks.

4.2 Task Network Generation

We now define how UCPARSE generates its search space via precondition achievement. As previously illustrated, UCPARSE begins with a null plan corresponding to the problem, and generates the successors of a given plan by iteratively selecting an open condition, nondeterministically choosing how to support it with a causal link, and nondeterministically choosing how to resolve interactions. While generating a new plan, UCPARSE tests it against the FHTN planning problem's coordinated set of tasks. See figure 4.2 for the formal definition of the precondition achievement half of our algorithm.

Given our formal definition of UCPARSE and a FHTN planning problem's charac-

Algorithm UCPARSE($\langle T, B, O, L\rangle, \langle T_*, S_*, D_*\rangle, agenda$)

1. **Termination:** If *agenda* has no selectable open conditions, fail if $D_* \neq \emptyset$; otherwise succeed and return $\texttt{sol}(\langle T, B, O, L\rangle)$.
2. **Open condition resolution:** Select $\langle Q_k@n_k\rangle$ (not referring to a member of D_*) in *agenda*, and resolve it with one of the two cases:
   (a) **Non-literal:** If Q_k is not a literal, nondeterministically choose a conjunct Γ from the set $\textsc{dnf}(Q_k, \mathcal{I})$, remove $\langle Q_k@n_k\rangle$ and add $\langle \ell@n_k\rangle$ to *agenda* for each literal $\ell \in \Gamma$, and add Γ's constraints to B– failing on inconsistent B;
   (b) **Literal:** If Q_k is a literal, either fail if $m \xrightarrow{\neg Q_k} n_k \in L$ (an impossible plan) or:
      i. **Parse:** use the steps in figure 4.8 to determine τ_n to support $\langle Q_k@n_k\rangle$;
      ii. **Satisfaction:** Add $(n \xrightarrow{Q_k} n_k)$ to L, add τ_n to T if new, and add $(n \prec n_k)$ to O when both τ_n and τ_{n_k} are primitive (fail if O becomes inconsistent).
      iii. **Subgoal generation:** Where τ_n executes a, alter *agenda* to remove $\langle Q_k@n_k\rangle$, add $\langle pre(a)@n\rangle$ when τ_n new, and add $\langle \Sigma^a_{Q_k}@n\rangle$.
3. **Causal link protection:** Primitive task τ_l *threatens* a causal link $n \xrightarrow{\ell} m$ when it asserts $\neg\ell$ between primitives τ_n and τ_m. For each link $n \xrightarrow{\ell} m \in L'$ and each (l:$\texttt{do}[a]$) that might threaten it, nondeterministically choose to:
   (a) **Promotion:** Add $(m \prec l)$ to O, and fail if inconsistent.
   (b) **Demotion:** Add $(l \prec n)$ to O, and fail if inconsistent.
   (c) **Confrontation:** Add $(n \prec l)$ and $(l \prec m)$ to O, and $\langle \neg\Sigma^a_{\neg\ell}@n\rangle$ to *agenda*.
4. **Recursive call:** Call UCPARSE($\langle T, B, O, L\rangle, \langle T_*, S_*, D_*\rangle, agenda$)

Figure 4.2: Algorithm solving FHTN problems via precondition achievement.

terization as a tuple $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$, we initially invoke UCPARSE with the plan

$$\langle T = \{\tau_0, \tau_\infty\}, B^{\emptyset}_{\mathtt{s}}, O^{\emptyset}_{\mathtt{s}}, L = \emptyset \rangle,$$

where $[\tau_{\mathtt{s}} \Rightarrow \langle \emptyset : T_{\mathtt{s}}, B_{\mathtt{s}}, O_{\mathtt{s}}, L_{\mathtt{s}} : \emptyset \rangle]$ is the distinguished method for decomposing $\tau_{\mathtt{s}}$. We also use the *agenda* to keep track of open goals, and assuring that the solution attains the problem's goal ρ involves making the *agenda* initially contain the single open goal $\langle \rho @ \infty \rangle$.

The main point where UCPARSE differs from UCPOP revolves around the parse line, where τ_n is determined; here UCPARSE uses a derivation $\langle T_*, S_*, D_* \rangle$ to match a plan against the problem's tasks $\mathcal{M}$, we delay describing this parameter until section 4.3's discussion on derivation maintenance.

4.2.1 Agenda Maintenance

In UCPARSE the *agenda* contains a partial plan's set of unresolved problems; this set is altered by four different lines: open condition resolution, non-literal resolution, subgoal generation, and confrontation. In open condition resolution, a problem is removed from the *agenda* and UCPARSE resolves it. This resolution may involve adding other problems to the agenda. If the selected problem is a non-literal goal, it gets broken into pieces in the non-literal resolution line, and the pieces are put back into the *agenda*. Eventually UCPARSE selects a literal open condition and satisfies it by adding a causal link to the plan. To assure that this causal link's property is attained, the subgoal generation line creates an open condition to add to the *agenda*. After assuring attainment, the confrontation line adds to the *agenda* to assure that tasks do not clobber the property while it is needed.

In order to concretely explain UCPARSE, we describe its behavior when solving our briefcase problem; solving this problem requires calling UCPARSE five times to move the briefcase, twice to leave the paycheck at home, five times to assure that moving the briefcase does not move the paycheck, and once to recognize a solution plan. For the initial call we pass in the null plan and an initial *agenda* computed from the problem's goal; this plan and agenda appear in figure 4.3.

$\boxed{0\text{:}\mathtt{do[start()]}}$ $\boxed{\infty\text{:}\mathtt{do[stop()]}}$

$$agenda : \{\langle (at(\mathtt{P},\mathtt{home}) \wedge at(\mathtt{B},\mathtt{office}))@\infty\rangle\}$$

Figure 4.3: Plan and agenda for UCPARSE's first call

Since the *agenda* is nonempty, control during the first call of UCPARSE passes to the open condition resolution line. Unlike other partial order planners, UCPARSE cannot choose any open condition here. Although we delay discussing derivation maintenance, we must describe how matching a plan with a derivation affects goal selection. There are times when we are temporarily unable to determine where a primitive task actually fits into the problem's tasks; when such a time occurs we delay reasoning about that task's preconditions until determining how it fits. We will see later that these tasks appear in D_*; thus the open condition resolution line can only select a precondition that does not refer to a task in D_*.

With respect to the first invocation, the *agenda* only has a single conjunctive goal to select. Upon selecting and removing the goal, UCPARSE discovers that Q_k is a conjunctive formula. Calling DNF$(Q_k, \mathcal{I})$ in the non-literal resolution line breaks Q_k down into literals by computing its disjunctive normal form. We defined DNF in the previous chapter as a function that computes an arbitrary closed formula's disjunctive normal form. The return value is a set containing sets of literals, where each set corresponds to a conjunct in the formula's disjunctive normal form. Γ in our example contains the literals are $at(\mathtt{P},\mathtt{home})$ and $at(\mathtt{B},\mathtt{office})$. After adding the literals' associated open goals to the *agenda*, control drops through to recursively make the second call with the altered agenda:

$$\{\langle at(\mathtt{P},\mathtt{home})@\infty\rangle, \langle at(\mathtt{B},\mathtt{office})@\infty\rangle\}$$

4.2.2 Precondition Support

On the second call, there are two open conditions in *agenda* and control passes to the open condition resolution line. While UCPARSE can select either open condition, we choose $\langle at(\mathtt{B}, \mathtt{office})@\infty\rangle$ to expose certain features of the algorithm. Given this choice, control passes to the literal subcase of open condition resolution.

The three lines within the literal subcase are devoted toward supporting a task's preconditions; task τ_n's precondition p is *supported* if there exists a causal link $m \xrightarrow{p} n$ and all preconditions required for task τ_m to assert p are supported. In partial order planning, supporting a precondition involves successively adding the link $m \xrightarrow{p} n$ and adding open goals to assure the eventual support of τ_m's relevant preconditions; UCPARSE also restricts how to support a precondition.

Continuing our briefcase example, the parse line consults the derivation to non-deterministically determine a new or existing task τ_n that can assert $at(\mathtt{B}, \mathtt{office})$; in this case, the only choice is a new (1.1:`do[carry(`$v_{1.1}$`,office)]`). Thus, the next line satisfies the open condition by adding: this task[1], an ordering constraint assuring that $\tau_{1.1}$ occurs in time, and a causal link assuring that the step's relevant effect persists until needed.

After satisfying the open condition, control passes to the subgoal generation line to assure that the precondition is eventually supported. This line adds the open goal

$$\langle \Big(pre(\mathtt{carry}(v_{1.1}, \mathtt{office})) \wedge \Sigma^{\mathtt{carry}(v_{1.1},\mathtt{office})}_{at(\mathtt{B},\mathtt{office})}\Big) @1.1\rangle.$$

While last chapter's definition of Σ^a_ℓ works just fine for grounded actions and literals, cases like this goal require us to generalize the definition to handle actions and literals that are only partially specified through the use of variables; the previous goal is such

[1] Notice that a new task's label and parameters are specified by the parse line. We shall explain how they are determined when explaining derivations.

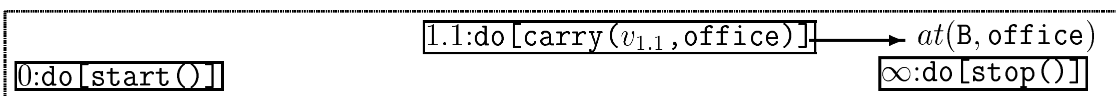
a case. For these cases we use the disjunctive equation

$$\Sigma_\ell^f = \bigvee_{\langle \rho_i, \ell_i \rangle \in \mathsf{eff}_f \wedge \ell = \ell_i \backslash b_i} (\rho_i \backslash b_i) \wedge b_i^e, \tag{4.1}$$

where b_i^e denotes those equality constraints in b_i that refer to arguments in $f(v_1, \ldots, v_k)$. Using this equation, the added goal becomes

$$\langle (v_{1.1} \neq \mathtt{office}) \wedge at(\mathtt{B}, v_{1.1}) \wedge ((TRUE \wedge (\mathtt{office} = \mathtt{office})) \vee$$
$$(in(\mathtt{B}) \wedge (\mathtt{B} \neq \mathtt{B}) \wedge (\mathtt{office} = \mathtt{office})))@1.1\rangle.$$

Finally, since no steps affect the newly added link, control passes through to UCPARSE's third call; the plan and agenda for this invocation appear in figure 4.4.



$$agenda : \{\langle (v_{1.1} \neq \mathtt{office}) \wedge at(\mathtt{B}, v_{1.1}) \wedge$$
$$((TRUE \wedge (\mathtt{office} = \mathtt{office})) \vee$$
$$(in(\mathtt{B}) \wedge (\mathtt{B} \neq \mathtt{B}) \wedge (\mathtt{office} = \mathtt{office})))@1.1\rangle,$$
$$\langle at(\mathtt{P}, \mathtt{home})@\infty\rangle\}$$

Figure 4.4: Plan and agenda for UCPARSE's third call

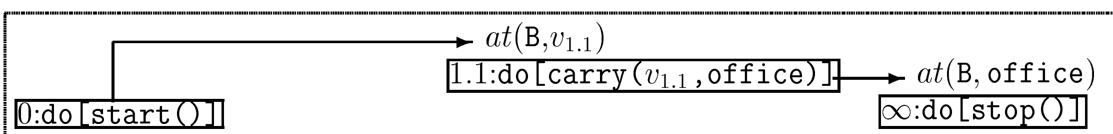
In many respects the third and fourth calls are similar to the first and second calls respectively. Once again UCPARSE breaks down a non-literal open condition and then supports a literal. The places where these calls differ stem from the complexity of the non-literal and how to support the literal.

During the third call, control passes to the open condition resolution line, where a non-literal goal is selected and broken down. One difference between this call and the initial call stems from DNF returning the two element set

$$\{\{(v_{1.1} \neq \mathtt{office}), at(\mathtt{B}, v_{1.1}), (\mathtt{office} = \mathtt{office})\},$$
$$\{(v_{1.1} \neq \mathtt{office}), at(\mathtt{B}, v_{1.1}), in(\mathtt{B}), (\mathtt{B} \neq \mathtt{B}), (\mathtt{office} = \mathtt{office})\}\}.$$

With this set the choice for Γ becomes nondeterministic, but in this case selecting the second set immediately leads to failure due to the inconsistent ($\mathtt{B} \neq \mathtt{B}$) constraint. In general, each element of Γ is either a literal or a binding constraint. While literals are turned into open goals for the agenda, binding constraints are immediately added to the plan's bindings. Backtracking occurs whenever the binding constraints become inconsistent. Thus UCPARSE, in this recursive call, must select the first set, add the two constraints to B, add $\langle at(\mathtt{B}, v_{1.1})@1.1\rangle$ to *agenda*, and then perform the fourth call.

In the fourth call, $\langle at(\mathtt{B}, v_{1.1})@1.1\rangle$ is selected and control passes to the literal subcase of open condition resolution. Supposing that the parse line chooses to use the initial conditions encoded in step τ_0's effects, performing the satisfaction and subgoal generation lines results in UCPARSE's fifth call with the plan and agenda shown in figure 4.5.



$agenda : \{\langle(TRUE \wedge (v_{1.1} = \mathtt{home}))@0\rangle, \langle at(\mathtt{P}, \mathtt{home})@\infty\rangle\}$

Figure 4.5: Plan and agenda for UCPARSE's fifth call

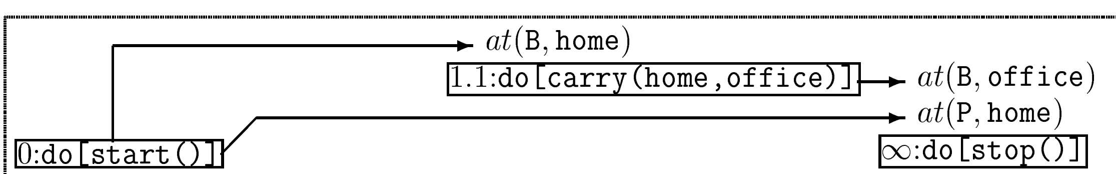
4.2.3 Interaction Resolution

As shown in figure 4.5, UCPARSE's fifth call has two open goals, and control passes to open condition resolution. Supposing that $\langle(TRUE \wedge (v_{1.1} = \mathtt{home}))@0\rangle$ is chosen, control passes to the non-literal subcase, where DNF returns $\{\{(v_{1.1} = \mathtt{home})\}\}$, and the constraint is added to B. At this point control passes to the sixth call.

Since the sixth call has one goal $\langle at(\mathtt{P}, \mathtt{home})@\infty\rangle$ left in the *agenda*, open condition resolution must select this goal then pass control to the literal subcase. Supposing that the parse line of the literal subcase decides to satisfy $\langle at(\texttt{P}, \texttt{home})@\infty\rangle$ by adding a link to the initial conditions, this new causal link is added to L by the satisfaction line, and subgoal generation adds the appropriate open condition $\langle TRUE@0\rangle$ to *agenda*.

At this point causal link protection becomes an issue due to an interaction between the `carry(home,office)` action, and the causal link requiring that the paycheck stay at home. Since moving the briefcase also moves its contents, we have to assure that it does not threaten the causal link by moving the paycheck. To protect against the threat, UCPARSE nondeterministically chooses between three techniques: promotion, demotion, and confrontation. While promotion tries to promote the causal link before the threatening action, demotion tries to demote the causal link after the threatening action. In this case, neither technique is applicable because the causal link stretches from the initial task τ_0 to the final task τ_∞; the ordering constraints in the satisfaction line assure that all added steps occur between these two steps.

Fortunately, confrontation can still be used; this technique involves adding a preservation precondition $\Pi_\ell^{a'}$ to a task $(o'{:}\texttt{do[do[}a'\texttt{]]})$ to disable an effect ℓ. As mentioned in the previous chapter, the preservation condition $\Pi_\ell^{a'}$ is identical to the negated causation condition $\neg\Sigma_{\neg\ell}^{a'}$; for our threat, this condition is $\neg in(\texttt{P})$. Since only one technique will resolve the threat, this line simply adds the open condition $\langle\neg in(\texttt{P})@1.1\rangle$ to the *agenda*, and the seventh call uses the plan and agenda shown in figure 4.6



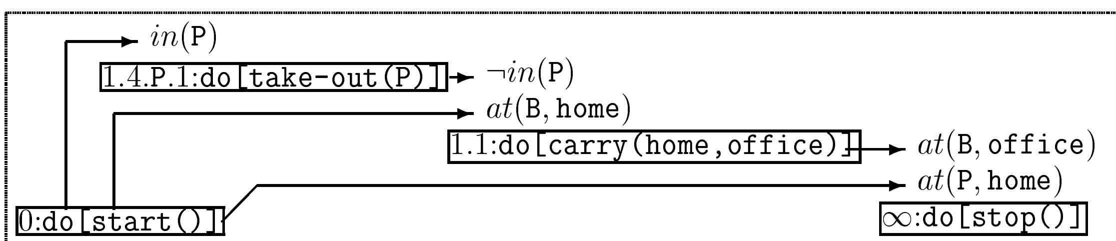
$$agenda : \{\langle TRUE@0\rangle, \langle(\neg in(\texttt{P}) \vee (\texttt{B} = \texttt{P}))@1.1\rangle\}$$

Figure 4.6: Plan and agenda for UCPARSE's seventh call

4.2.4 Termination

On the seventh call the non-literal precondition $\langle TRUE@0\rangle$ is resolved by DNF returning $\{\{\}\}$. Thus the non-literal subcase of open condition resolution need not add anything to B or *agenda* and control passes on to the eight call to resolve the other open condition.

Just as the first through fifth calls to UCPARSE breakup a non-literal open condition and then alters the plan to completely support $\langle at(\texttt{B}, \texttt{office})@\infty\rangle$, the eighth through twelfth calls breakup a non-literal and then completely support $\langle \neg in(\texttt{P})@1.1\rangle$. Since these five calls respectively mimic the first five calls, we can omit discussing them. Thus, on the thirteenth invocation, UCPARSE is passed the plan and agenda shown in figure 4.7.



agenda : $\{\}$

Figure 4.7: The final plan that solves our example problem

Since the thirteenth call's *agenda* is empty, control stops at line 1. Unlike partial order planners, UCPARSE cannot assume that a plan with no open conditions solves the problem. Although the plan might attain the problem's goal ρ, it might not perform the problem's tasks; some of the actions might not match tasks in the derivation. Since these tasks appear in D_*, a plan is a solution when D_* is empty. When *agenda* is empty and D_* has tasks UCPARSE must backtrack in search of another plan. As we will later see, the thirteenth call's plan does match the tasks, and solves the problem.

4.3 Incremental Parsing

While UCPARSE builds a frugal plan via precondition achievement, it maintains a derivation denoting the correspondence between each added primitive task and the problem's coordinated tasks. By enforcing the correspondence at each point during the planning process, large branches of the search tree are pruned; these branches might lead to solutions to the partial order problem, but they do not solve the FHTN problem. Thus the decomposition methods are used as a form of search control to make the planner avoid considering plans with unwanted behaviors.

As an FHTN problem's tasks and decomposition methods define a language of task networks, enforcing a correspondence between a partial plan and the problem's tasks is analogous parsing the prefix to a word as it is being written. Where parsing a prefix determines when a word cannot be extended into a member of the language, so does parsing a partial plan. To be more precise, each legal plan in UCPARSE's search space has a derivation denoting how to decompose the problem task τ_{s} into the plan.

Definition 10 *A* DERIVATION *is a tuple* $\langle T_*, S_*, D_* \rangle$, *where* T_* *denotes a set of instantiated task decompositions,* S_* *is a symbol table denoting a correspondence from each point protection to the primitive task that supports it, and* D_* *denotes those primitive tasks that have yet to be matched against the problem's tasks.*

The difference between UCPARSE and our prefix parsing analogy involves using the grammar to determine how to extend a plan. Instead of a generating a new plan and then trying to extend the derivation to match it, UCPARSE extends the derivation to determine what new plan to generate. In the previous section, we hid this extension process behind the parse line in the literal case of open condition resolution; we now expose this process in figure 4.8.

The main source of complexity in the parse line's substeps stem from instantiating elements from $\mathcal{M}$ instead of $G(\mathcal{M})$. To deal with variablized derivations, we introduce

To determine primitive τ_n for supporting a literal open condition $\langle Q_k@n_k\rangle$ do the following while backtracking whenever constraints B or O become inconsistent:

1. **Derivation traversal:** While ascending and descending, let $e \overset{b}{\in} S$ denote that there exists $e' \in S$ and $e \overset{B}{=} e' \backslash b$, where b denotes a substitution for universal variables.
   (a) **Ascend:** Starting with $n = n_k$, repetitively let $n = a(n)$ while $(\xrightarrow{Q_k} n) \overset{b}{\in} P_{a(n)}$ in T_*. Ultimately let $n = m$, where $(m \xrightarrow{Q_k} n) \overset{b}{\in} L_{a(n)}$ in T_*.
   (b) **Descend:** Repetitively let $n = m$ while $(m \xrightarrow{Q_k}) \overset{b}{\in} E_n$ in T_*.
2. **Universal variable resolution:** If n contains variables and $(T \cup S_*)$ lacks τ_n and (n', n), take all $[\tau_m \Rightarrow ctn_m]$, τ_m, or (m', m) from $(T_* \cup T \cup S_*)$, where $m \overset{b}{\cong} n$, and nondeterministically choose one or none. For those cases where one was chosen, add b to B and resume descend; Otherwise, add $O^n_{a(n)}$ to O and for each element add $m \ncong n$ to B.
3. **Reduction:** While n denotes abstract task τ_n: nondeterministically choose/add a new (instantiated from $\mathcal{M}$) method $[\tau_n \Rightarrow ctn_n]$ to T_*; find $(m \xrightarrow{Q_k}) \overset{b}{\in} E_n$; add B_n to B; add $O^{\emptyset}_n \cup O^m_n$ to O; and finally let $n = m$.
4. **Point protection:** Handle point protections by:
   (a) if $(m, n) \in S_*$, then let $n = m$;
   (b) if n is a new point protection, nondeterministically choose $\tau_m = (m{:}\texttt{do[}a\texttt{]})$ from a set of existing (from T) or new (instantiated from $\mathcal{O}$) tasks that can assert Q_k, add (m, n) to S_*, add τ_m to D_* if τ_m is new, and finally let $n = m$;
   (c) if n is a task label denoting a new task $(n{:}\texttt{do[}a\texttt{]})$, nondeterministically choose $(n'{:}\texttt{do[}a'\texttt{]}) \in D_* \cup \{\tau_n\}$ such that $a \overset{b}{\cong} a'$. Let $B = B \cup b$, remove $(n'{:}\texttt{do[}a'\texttt{]})$ from D_*, and substitute n for n' in T, O, L, S_*, D_*, and *agenda*.

Figure 4.8: The steps to perform derivation maintenance in UCPARSE

several notational conventions. While $m \overset{b}{\cong} n$ denotes the existence of a variable substitution b such that $m = n \backslash b$, $e \overset{b}{\in} S$ denotes $e = e' \backslash b$ for some element e' of set S. With regard to sets whose elements contain universal variables, $S^{\emptyset}$ and S^{n} respectively denote those elements of S that always lack universal variables and that refer to a label n with universal variables.

When initiating derivation maintenance, $\langle T_*, S_*, D_* \rangle$ denotes the derivation for the current plan, and $\langle Q_k @ n_k \rangle$ denotes the literal open condition to be resolved. The whole point is to assure that for any plan $\langle T, B, O, L \rangle$ there exists T', B', O' and L' such that $\tau_{\mathtt{s}} \overset{*}{\Rightarrow} \langle \emptyset : T', B', O', L' : \emptyset \rangle$, where $T' = T - D_*$, $B' \subseteq B$, $O' \subseteq O$, and $L' \backslash S_* = L$ whenever D_* is empty. For example, consider the null plan $\langle T, B, O, L \rangle$'s initial derivation

$$\langle T_* = \{[\tau_{\mathtt{s}} \Rightarrow \langle \emptyset : T_{\mathtt{s}}, B_{\mathtt{s}}, O_{\mathtt{s}}, L_{\mathtt{s}} : \emptyset \rangle]\}, S_* = \emptyset, D_* = \emptyset \rangle,$$

where T_* contains the unique decomposition in $\mathcal{M}$ for defining the start symbol $\tau_{\mathtt{s}}$ with the problem's coordinated set of tasks. Since $\langle T_{\mathtt{s}}, B_{\mathtt{s}}, O_{\mathtt{s}}, L_{\mathtt{s}} \rangle$ denotes the coordinated set of tasks to perform, the null plan's binding and ordering constraints are $B_{\mathtt{s}}^{\emptyset}$ and $O_{\mathtt{s}}^{\emptyset}$ respectively, and both τ_0 and τ_∞ are members of $T_{\mathtt{s}}$.

To explain derivation maintenance, we trace its behavior at representative points while UCPARSE solves our example briefcase problem. We simplify our explanations by graphically representing a derivation; While required tasks appear as boxes, the names of primitive and non-primitive tasks respectively appear inside and above their associated boxes, and subtask relationships appear as box containment. For instance, figure 4.9 represents the null plan's derivation; as start and stop primitives are subtasks of $\tau_{\mathtt{s}}$, they appear inside the box for $\tau_{\mathtt{s}}$.

4.3.1 Finding New Tasks

As mentioned earlier, UCPARSE adds (1.1:`do[carry(`$v_{1.1}$`, office)]`) to the null plan during its second call. When UCPARSE performs the parse line, the objective is to

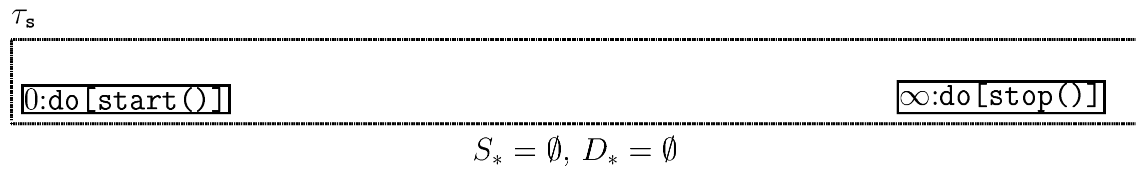


Figure 4.9: The null plan and its derivation

extend the null plan to support $\langle at(\mathtt{B}, \mathtt{office})@\infty \rangle$. From a derivation's perspective, the resultant plan looks like figure 4.10, where the label "1.1" comes from the subtask $\tau_{1.1}$ of `move-to` in $T_{\mathtt{s}}$.

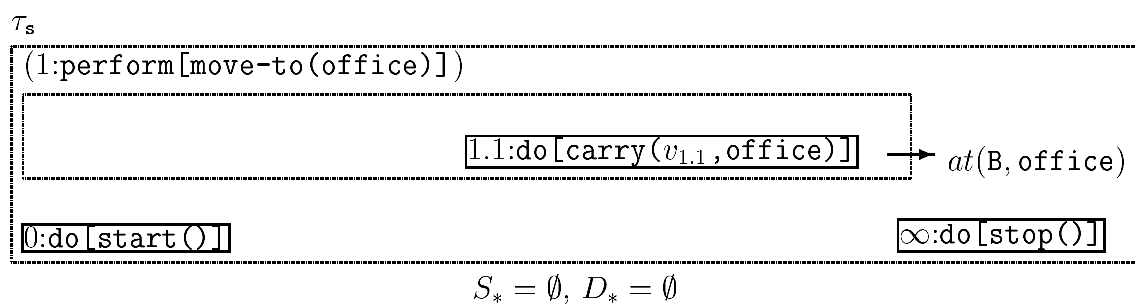


Figure 4.10: Plan and derivation after supporting $\langle at(\mathtt{B}, \mathtt{office})@\infty \rangle$ with $\tau_{1.1}$

During derivation maintenance, n denotes a variable used to trace auxiliary constraints through the derivation tree and find the task or point protection responsible for supporting an open condition $\langle Q_k@n_k \rangle$. The ascend segment of derivation traversal starts by determining how Q_k is asserted for τ_n. While τ_n is a subtask of $\tau_{a(n)}$ with $(\xrightarrow{\ell} n)$ in $P_{a(n)}$, n is altered to point to $\tau_{a(n)}$. In this way, the precondition is traced up through the parent tasks until finding m denoting τ_m, or a point protection, responsible for attaining Q_k; n is ultimately altered to denote the responsible element. In our example, τ_∞ is a subtask of $\tau_{\mathtt{s}}$ and ascending for $at(\mathtt{B}, \mathtt{office})$ immediately finds $(1 \xrightarrow{at(\mathtt{B},\mathtt{office})} \infty)$ in $L_{\mathtt{s}}$, where τ_1 performs a `move-to`; n ultimately points to this task.

Since the derivation lacks a decomposition method for τ_1, control drops straight to the reduction line. We delay discussing universal variable resolution until a later point; for now it suffices to say that unless variables or problem constants, like `P`, appear in n, control passes straight through to reduction. Since n contains the simple label "1", such is the case here.

Given that τ_1 is a non-primitive `perform` task, the reduction line nondeterministically chooses a method $[\tau_n \Rightarrow ctn_n]$ from $\mathcal{M}$ and instantiates it by unifying τ_1 with τ_n, substituting the variables in ctn_n with the unification results and making all unsubstituted variables unique. For instance, the non-primitive task τ_1 is $(1{:}\texttt{perform[move-to(office)]})$ and the only relevant decomposition method decomposes $(n{:}\texttt{perform[move-to(}w\texttt{)]})$; thus n and w in the method are respectively replaced with 1 and `office` while the other variables are made unique; the resultant method is

$$
\begin{aligned}
&(1 : \texttt{perform[move-to(office)]}) \Rightarrow \\
&\quad \langle P = \emptyset : \\
&\quad T = \{(1.1{:}\texttt{do[carry(}v_{1.1}\texttt{,office)]}), (1.3.\mathbf{c}{:}\texttt{achieve[}\neg in(\mathbf{c})\texttt{]}), \\
&\qquad\quad (1.4.\mathbf{b}{:}\texttt{achieve[}\neg in(\mathbf{b})\texttt{]})\}, \\
&\quad B = \emptyset, \\
&\quad O = \{(1.0 \prec 1.1), (1.1 \prec 1.\infty), \\
&\qquad\quad (1.0 \prec 1.3.\mathbf{c}.0), (1.3.\mathbf{c}.0 \prec 1.3.\mathbf{c}.\infty), (1.3.\mathbf{c}.\infty \prec 1.\infty), \\
&\qquad\quad (1.0 \prec 1.4.\mathbf{b}.0), (1.4.\mathbf{b}.0 \prec 1.4.\mathbf{b}.\infty), (1.4.\mathbf{b}.\infty \prec 1.\infty)\}, \\
&\quad L = \{(1.2 \overset{at(\texttt{B},v_{1.1})}{\rightarrow} 1.1), (1.4.\mathbf{b} \overset{\neg in(\mathbf{b})}{\rightarrow} 1.1), (1.3.\mathbf{c} \overset{in(\mathbf{c})}{\rightarrow} 1.1)\} : \\
&\quad E = \{(1.1 \overset{at(\mathbf{a},\texttt{office})}{\rightarrow})\}\rangle.
\end{aligned}
$$

Upon adding the new method to T_*, reduction adjusts the binding and ordering constraints by adding sets $B_1^\emptyset$ and $O_1^\emptyset$ respectively. Thus B stays empty while O becomes the set

$$\{(0 \prec 1.0), (1.0 \prec 1.\infty), (1.\infty \prec \infty), (1.0 \prec 1.1), (1.1 \prec 1.\infty)\}.$$

After updating the derivation and constraints, n is updated to reflect the subtask that causes the requested condition. In this case n becomes 1.1 and $\tau_{1.1}$ provides the requested condition; since this task is primitive, control now passes to the point protection line.

Upon reaching point protection, derivation maintenance has completely traced the derivation's auxiliary constraints to determine the primitive task, or point protection, that provides the desired effect. Since τ_n is a new task and D_* is empty, the point protection line does effectively nothing, and we will delay discussing it. Control at this point passes back to the precondition achievement half of UCPARSE.

4.3.2 Finding New Point Protections

The next extension to the derivation in our example trace occurs during UCPARSE's fourth call. Now the parse line's objective is to extend the plan in order to resolve $\langle at(\texttt{B}, v_{1.1})@1.1\rangle$, and the resultant derivation looks like figure 4.11. The main point is that derivation maintenance now finds a new point protection $(1.2 \overset{at(\texttt{B},v_{1.1})}{\rightarrow})$ and uses it as a directive to link to the initial conditions.

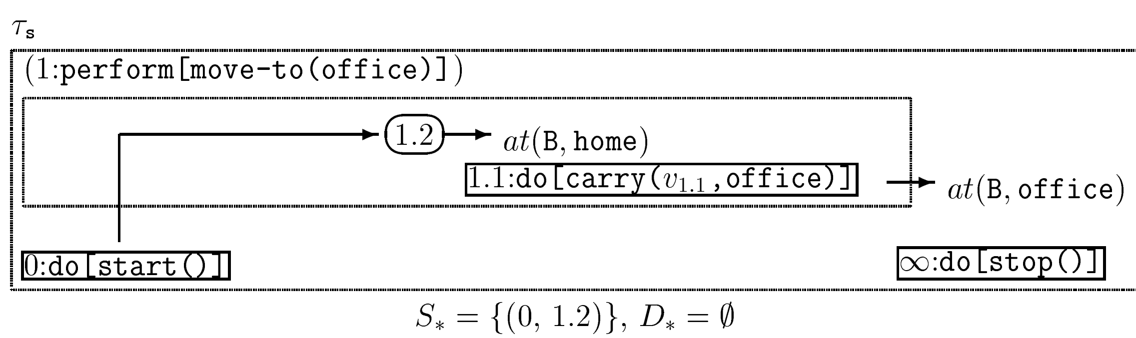


Figure 4.11: Plan and derivation after supporting $\langle at(\texttt{B}, v_{1.1})@1.1\rangle$ with τ_0

Just as before, the first three lines of derivation maintenance use n to trace through the derivation tree to find the task or point protection responsible for attaining the selected open condition. This time the ascend segment immediately finds $1.2 \overset{at(\texttt{B},v_{1.1})}{\rightarrow} 1.1$

in L_1 and n becomes 1.2. Since there is no $\tau_{1.2}$ in T_1, and 1.2 contains no variables, control passes straight through to the point protection line.

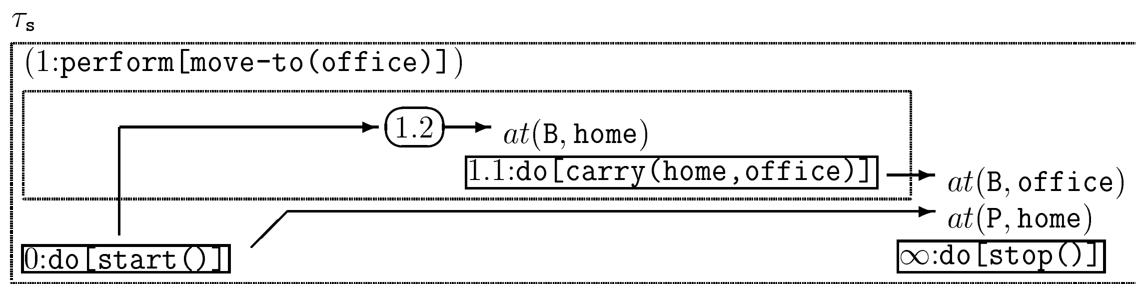
In FHTN planning, a point protection is satisfied by an assigned supporting primitive task whose effect persists until needed by the point protection; this assigned task must have an another reason for existing in the plan. During the planning process point protections can be in one of three states: unsupported, supported by an unjustified task, and supported by a justified task. A task is justified when it has a reason, besides supporting point protections, for being in the plan.

The point protection line handles point protections by using one of three cases. The first involves using S_* to assure that a unique single task supports a point protection, even if derivation maintenance runs into the point protection in more than one call to UCPARSE. The second case involves either reusing an existing primitive task or adding a new one to support the protection. A task added at this point is unjustified because its sole purpose is to support the protection, and D_* denotes the set of unjustified tasks. This brings us to the third case, where unjustified tasks can get justified and removed from D_*. Whenever derivation maintenance traces auxiliary constraints to discover that a new primitive task must be added to the plan, it has the option of either adding the task or justifying one in D_*.

In our example S_* is initially empty and n denotes an unsupported point protection; so the second case applies. Since there is a nondeterministic choice here, UCPARSE can pursue one of several options. Here the options are to either use τ_0 or add an unjustified primitive task to `carry` the briefcase to location $v_{1.1}$. Choosing the first option gives us our illustrated result where S_* now enforces the requirement that τ_0 support protection 1.2. While choosing the second option is valid here, it eventually leads to failure due to an inability to justify the `carry` task.

4.3.3 Finding Existing Features

While the last two examples of derivation maintenance had to extend a derivation in order to use it, UCPARSE during its sixth call just refers to the derivation to determine how to modify the plan to support $\langle at(\texttt{P}, \texttt{home})@\infty\rangle$. In this case the ascend line immediately finds $0 \overset{at(\texttt{P},\texttt{home})}{\rightarrow} \infty$ in $L_{\texttt{s}}$. Since τ_0 is an existing primitive task, control passes out of the parse and we get the plan in figure 4.12.



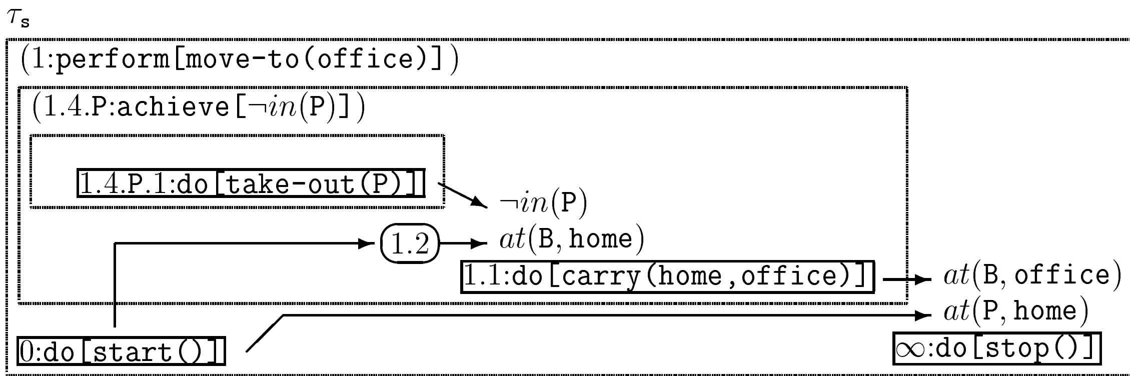
$S_* = \{(0,\ 1.2)\}$, $D_* = \emptyset$

Figure 4.12: Plan and derivation after supporting $\langle at(\texttt{P}, \texttt{home})@\infty\rangle$ with τ_0

We now take advantage of the simplicity of this invocation to expand our discussion to showing how tasks help UCPARSE prune its search space. Instead of linking to the initial conditions, UCPOP might have tried to satisfy the goal $\langle at(\texttt{P}, \texttt{home})@\infty\rangle$ by adding a `carry` action to carry the paycheck home. Derivation maintenance avoids this hypothetical situation by never suggesting to add a `carry` task.

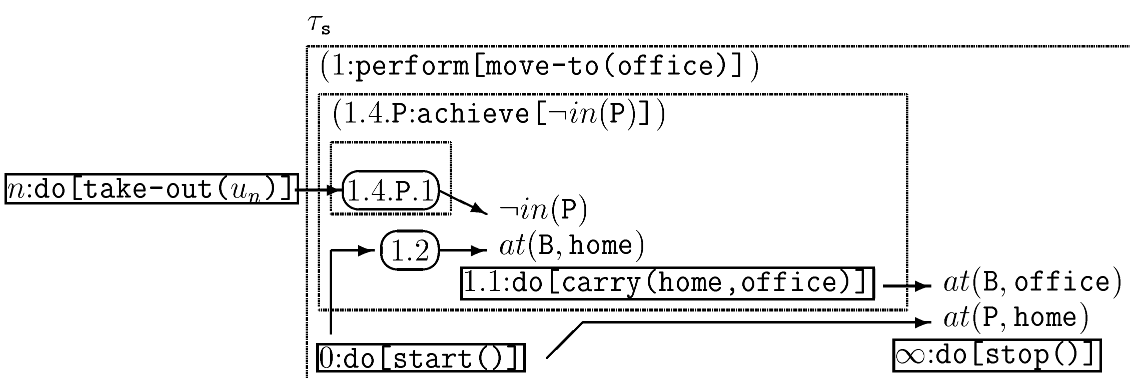
4.3.4 Multiple Derivations

Although the seventh through twelfth invocations of UCPARSE are not interesting from precondition achievement's perspective, the ninth call does illustrate interesting features of derivation maintenance. In this call UCPARSE determines how to take the paycheck out of the briefcase in order to leave it at home. With respect to our formalism, UCPARSE is now supporting $\langle \neg in(\texttt{P})@1.1\rangle$, and the result of derivation maintenance looks like one of the two derivations in figures 4.13 and 4.14. One point of interest stems from the very fact that the same plan can be suggested by more than one derivation. This suggests that more than one branch in the search space can contain the same plan $\langle T, B, O, L\rangle$, but the plan must be extended in different ways in accordance with the different derivations.



$S_* = \{(0,\ 1.2)\},\ D_* = \emptyset$

Figure 4.13: Plan and derivation after supporting $\langle\neg in(\mathtt{P})@1.1\rangle$ with $\tau_{1.4.\mathtt{P}.1}$



$S_* = \{(0,\ 1.2), (\mathrm{n},\ 1.4.\mathtt{P}.1)\},\ D_* = \{\tau_n\}$

Figure 4.14: Derivation after supporting $\langle\neg in(\mathtt{P})@1.1\rangle$ with point protection 1.4.P.1

To generate these derivations, the incremental parse step starts by traversing the derivation from figure 4.12. In this case, the ascent discovers $(1.4.\mathtt{P} \overset{\neg in(\mathtt{P})}{\rightarrow} 1.1) \overset{b}{\in} L_1$, where b is the substitution $\{(\mathbf{b}, \mathtt{P})\}$. Since this is the first case where b is not empty, it is also the first case where $O^b_{a(n)}$ is not empty; O is unioned with the set

$$\{(1.0 \prec 1.4.\mathtt{P}), (1.4.\mathtt{P} \prec 1.\infty)\}.$$

Since $\tau_{1.4.\mathtt{P}}$ denotes the nonprimitive $(1.4.\mathtt{P}{:}\mathtt{achieve[}\neg in(\mathtt{P})\mathtt{]})$ with no decomposition method, control passes to the reduction line. By nondeterministically choosing a decomposition method, this line can follow two options that lead to the two derivations. While the first option leads to figure 4.13 by choosing to decompose $\tau_{1.4.\mathtt{P}}$ as a `perform[unload(P)]` task, the second option leads to figure 4.14 by choosing to phantomize the achieve task.

While the options add different chosen decompositions to T_*, they both find $(1.4.\mathtt{P}.1 \overset{\neg in(\mathtt{P})}{\rightarrow})$ in $E_{1.4.\mathtt{P}}$; 1.4.P.1 denotes either a primitive `take-out` task or a point protection depending on the choice. If the label denotes a primitive task, the point protection line's third case has no effect due to the lack of elements in D_* and figure 4.13 is the ultimate result. In the alternative option control passes to the second case of derivation because 1.4.P.1 denotes a new point protection. Since no existing task will assert $\neg in(\mathtt{P})$, a new task must be added, and the only task that can assert $\neg in(\mathtt{P})$ is a primitive `take-out` task. Thus the second case has a single option to add an unjustified `take-out` task.

Although the derived label in our example contained no variables, such is not the case in general. Suppose that the `carry` step acquired the precondition $in(x)$, where x is an unbound variable; tracing the derivation's auxiliary constraints yields the task label $1.3.x$, and ultimately a decomposition for task $(1.3.x{:}\mathtt{achieve[}in(x)\mathtt{]})$ would find its way into T_*. Assuming that all variables appearing in a solution plan are bound to constants, the UCPARSE will eventually bind x in the label to a constant. There is a danger here that two tasks might eventually acquire the same label while adding variable constraints. We avoid this problem by adding $m \ncong n$ constraints to

B during universal variable resolution.

4.4 Summary

In this chapter we presented UCPARSE an algorithm based on precondition achievement that solves FHTN problems. The difference between UCPARSE and partial order planners, like UCPOP, stems from using a task-network grammar to determine which action achieves a precondition; UCPOP will try all possible actions. While UCPARSE does not try all possible actions, its search space need not be smaller than UCPOP's. UCPARSE searches a space of (plan, derivation) tuples, where a derivation denotes a possible way to parse a plan in accordance with the grammar; if the grammar is ambiguous, there can be cases where the same plan appears multiple times in the search space with different derivations.

To illustrate how UCPARSE generates its search space, we traced its behavior while solving our example briefcase problem. Even this problem exhibited a certain amount of ambiguity. The problem was solved within thirteen recursive calls, where the search space branched in the fourth and ninth calls. In the ninth call, both suggested branches in the search space used the same plan, but the derivations were different. These calls are summarized in the following items, and the derivation for the solution plan is illustrated in figure 4.15.

1. Break $\langle(at(\mathtt{P},\mathtt{home}) \wedge at(\mathtt{B},\mathtt{office}))@\infty\rangle$ up into its components.

2. Add (1.1:`do[carry(`$v_{1.1}$`,office)]`) to support $\langle at(\mathtt{B},\mathtt{office})@\infty\rangle$.

3. Break $\langle(v_{1.1} \neq \mathtt{office}) \wedge at(\mathtt{B}, v_{1.1}) \wedge ((TRUE \wedge (\mathtt{office} = \mathtt{office})) \vee (in(\mathtt{B}) \wedge (\mathtt{B} \neq \mathtt{B}) \wedge (\mathtt{office} = \mathtt{office})))@1.1\rangle$ up into its components.

4. Use initial conditions to support $\langle at(\mathtt{B}, v_{1.1})@1.1\rangle$.

5. Resolve $\langle TRUE \wedge (v_{1.1} = \mathtt{home})@0\rangle$ by adding the binding constraint.

6. Use initial conditions to support $\langle at(\mathtt{P}, \mathtt{home})@\infty\rangle$; `carry` action conflicts.

7. Resolve $\langle TRUE@0\rangle$; this goal evaporates.

8. Break $\langle(\neg in(\mathtt{P}) \vee (\mathtt{P} = \mathtt{B}))@1.1\rangle$ up into its components.

9. Add (1.4.P.1:`do[take-out(P)]`) to support $\langle\neg in(\mathtt{P})@1.1\rangle$.

10. Break $\langle in(\mathtt{P}) \wedge TRUE \wedge (\mathtt{P} = \mathtt{P})@1.4.\mathtt{P}.1\rangle$ up into its components.

11. Use initial conditions to support $\langle in(\mathtt{P})@1.4.\mathtt{P}.1\rangle$.

12. Resolve $\langle TRUE@0\rangle$; this goal evaporates.

13. Identify and return solution in figure 4.15.

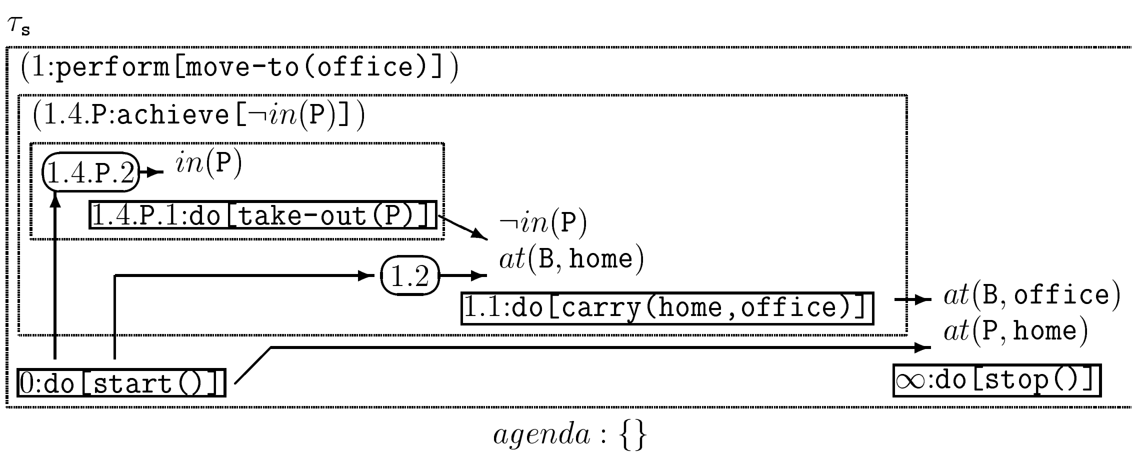


Figure 4.15: Derivation of the solution task network.

Chapter 5

THE UCREDUCE ALGORITHM

I this chapter we describe the UCREDUCE algorithm for solving FHTN planning problems. In the previous chapter we discussed UCPARSE in terms of using a problem's goals to drive plan synthesis and using the problem's tasks to limit it; UCREDUCE takes the complementary approach by using a problem's tasks to drive plan synthesis and using goals to limit it. In the following three sections we show: how a set of tasks and decomposition methods can drive plan synthesis via reduction; how UCREDUCE uses reduction; and how UCREDUCE uses goals and frugality to limit plan generation.

5.1 Reduction

Typical HTN planners take a task network and repeat a two step process where they (1) use a decomposition method to expand a task into a network of subtasks and (2) use a *critic function* to resolve interactions. This cycle begins with a given problem task network, and continues until only primitive tasks remain or the critic reaches a point where it cannot resolve an interaction (in which case backtracking is required).

In HTN planning, the relationships between the tasks and subtasks within a hierarchical task network are often illustrated using a tree, like the one in figure 5.1. Each of the tree's nodes represents a task, and a task's subtasks determine a node's children; for instance, figure 5.1's nodes represent tasks within the solution of our example problem. Sitting at the top is a node for task $\tau_{\mathtt{s}}$, representing the problem's initial task network. Since the initial task network has three tasks, the top node has three children, one of which is non-primitive. This non-primitive `perform[]` task has two children to carry the briefcase to the office, and assure that the paycheck remains at home. Finally, the non-primitive `achieve[]` subtask uses a `take-out` action to attain its goal.

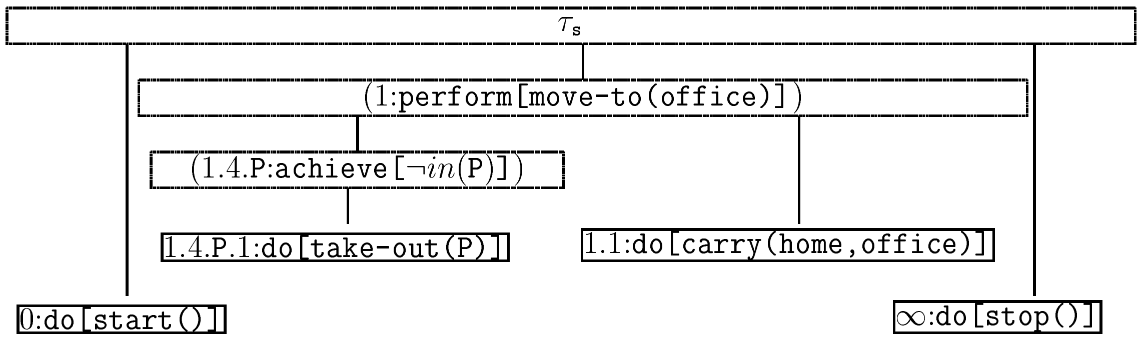


Figure 5.1: Derivation tree for our example problem's solution

During planning, the tree's leaves correspond to tasks within the current partial plan; reduction changes these leaves by adding children to a selected leaf in accordance with a selected reduction method. As such, planning continues until all leaves denote primitive tasks or phantomized achieve tasks[1]. Assuming that the critic function appropriately orders the final tasks to facilitate constructive interactions while avoiding destructive interactions, the leaves in the final tree correspond to a solution of the problem; such is the case for figure 5.1.

5.1.1 Adapting a Classical HTN Planner

One problem with the common definition of an HTN planner involves the assumption that either those tasks initially provided in the problem do not interact or one of the tasks is non-primitive; violating this assumption leads to unsoundness. For instance, suppose that the objective of a problem is to take a set of primitive tasks and schedule

[1] The special case for these achieve tasks arises from the fact that a phantomizing reduction method does not add any tasks.

their execution to avoid interactions. In this case, the classical algorithm definition immediately terminates due to the absence of non-primitive tasks, and the problem's original task network is returned; this solution is not sound because the original task network might contain destructive interactions between its tasks. A final repair function can fix this problem; it just has to perform an extra interaction resolution.

As figure 5.2 shows, a simpler way to drop the common assumption and still maintain completeness involves reformulating the reduction algorithm to place the critic function first; in doing this we assure that unwanted interactions in the initial task network are always resolved. Notice that the critic function is still called after each reduction; the only difference is that now the critic is also called prior to making the first termination test.

Algorithm: HTN-PLAN(tn)

1. **Interaction resolution:** Nondeterministically choose $tn' \in \text{CRITIC}(tn)$.
2. **Termination:** Return `sol(`tn'`)` if tn' contains no non-primitive tasks.
3. **Task selection:** Let τ_n be a non-primitive task in tn'.
4. **Reduction:** Nondeterministically choose a method $m \in \mathcal{M}$ that expands τ_n, and let $tn'' = \text{REDUCE}(tn', \tau_n, m)$.
5. **Recursive invocation:** Call HTN-PLAN(tn'').

Figure 5.2: Revised Basic HTN planning algorithm

As we shall later see, the UCREDUCE algorithm is an adaptation of our revised basic HTN planning algorithm, the differences arise from the extra frugality requirement of FHTN problems. In FHTN planning, a task's subtasks depend on desired effects and unwanted side-effects; some of which are determined as interactions detected by the critic function. Since some of a task's interactions might be detected after the task is decomposed, reduction by replacing a task with its subtasks becomes infeasible; UCREDUCE may not know which subtasks to include.

In this chapter, we show how UCREDUCE breaks up the reduction phase by tracing a method's causal structure and only adding those subtasks required to attain the expanded task's effects. As interactions are detected, a task's decomposition method gets revisited, and more subtasks are added. Intuitively, from the derivation tree's perspective, expanding a task involves associating a decomposition method with the task and adding the currently required subtasks as children. The task's associated method determines what new subtasks to add when they become required; such requirements computed by tracing auxiliary constraints through the derivation tree as required preconditions are detected.

5.1.2 Modeling Tasks With ADL

In order to facilitate tracing a plan's causal structure when it contains abstract tasks, we model these tasks as actions. In HTN planning, tasks are typically modeled using STRIPS operators; such operators are expressive enough when a task's subtasks depend solely on decomposition methods and each action has a constant set of effects. In our planners we drop both of these requirements; both the number of a task's subtasks and effects of an action depend on context. For this reason, we use ADL operators to model abstract tasks as well as actions.

To illustrate our modeling abstract tasks, consider our example problem's three `perform[]` tasks: `move-to(`w`)` moves a set of objects to a location w; `load(`u`)` gets an object u into the briefcase; and `unload(`u`)` removes it. As mentioned earlier each task is defined using methods of the form

$$\tau_n \Rightarrow \langle P_n : tn_n : E_n \rangle,$$

where tn_n is the coordinated set of subtasks, E_n denotes how these subtasks produce τ_n's requested effects, and P_n denotes how tn_n uses τ_n's required preconditions.

To model a task τ_n with an ADL action, we determine both effects and those preconditions needed to produce each particular effect. Determining a task τ_n's possible effects is just a matter of inspecting the E_n sets of its related decomposition methods. Assuming that the domain is well formed, the literals mentioned in each E_n set are identical; the literals obtained by unioning the literals from all relevant E_n sets denote the task's possible effects. Similarly, τ_n's preconditions are related to the P_n sets of τ_n's relevant decomposition methods. Since all P_n sets in our example problem are empty, none of the abstract tasks have preconditions; thus the three relevant ADL operators appear in table 5.1.

`perform[unload(`u`)]`	pre :	$TRUE$
	eff :	$\langle TRUE, \neg in(u)\rangle$
`perform[load(`u`)]`	pre :	$(u \neq \texttt{B})$
	eff :	$\langle TRUE, in(u)\rangle$
`perform[move-to(`w`)]`	pre :	$TRUE$
	eff :	$\langle TRUE, at(u, w)\rangle$

Table 5.1: Action templates for abstract tasks

In order to provide a more interesting example of modeling a perform task with an ADL operator, suppose that we replaced `do[carry(`v, w`)]` with an abstract `perform[carry-path(`v, w`)]` task in order to add a route planning component to our problem. Essentially, each `perform[carry-path(`v, w`)]` denotes a sequence of `do[carry-step(`v_i, w_i`)]` tasks, where v_1 and w_n respectively denote the initial location v and ultimate destination w; in the sequence, w_i denotes the same intermediate location as v_{i+1}, and location v_i is next to w_i. We define these tasks with the two decomposition methods in figure 5.3. While figure 5.3a's method applies when v and w are connected, figure 5.3b's method uses `carry-path` to move to a location adjacent to w and `carry-step` to make the last step.

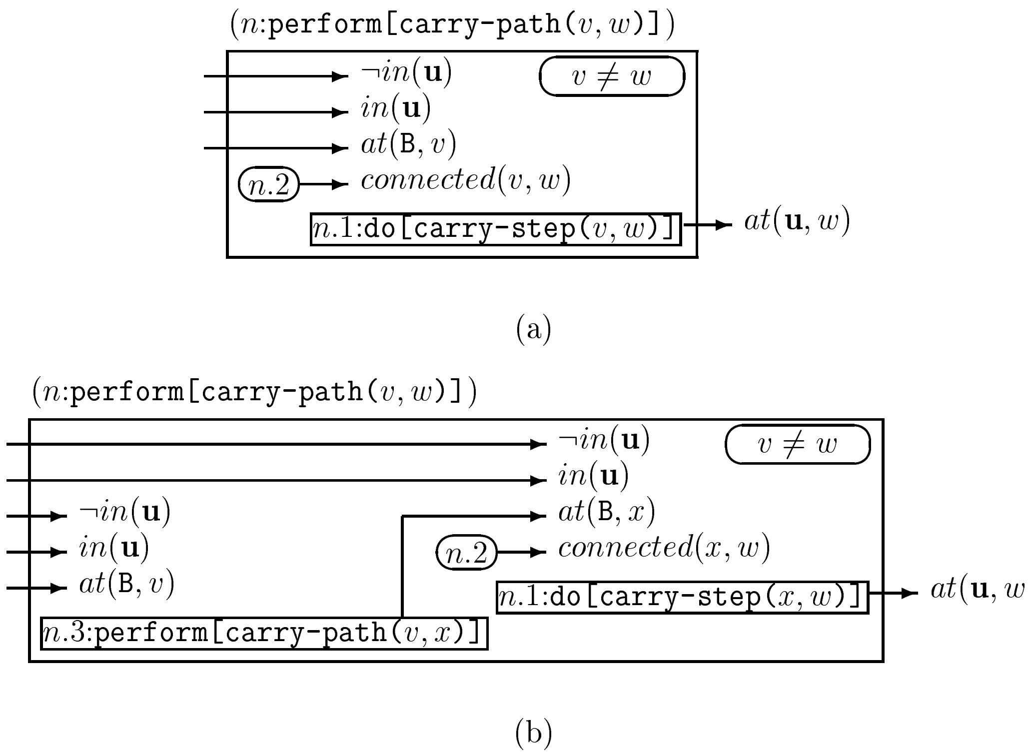


Figure 5.3: The two methods for defining (n:`perform[carry-path(`v, w`)]`)

With this definition of `carry-path`, route planning becomes a sub-problem of each problem in our original briefcase world. As such, planning occurs at two levels of abstraction; the top level involves moving objects by managing the briefcase's contents while carrying it between locations, and the bottom level involves reasoning about how to travel from one location to the next. To complete our definition of this refinement to `carry`, we define `carry-step` with the operator

`carry-step(`v, w`)`	pre :	$(v \neq w) \wedge at(\mathtt{B}, v) \wedge connected(v, w)$
	eff :	$\langle TRUE, at(\mathtt{B}, w)\rangle, \langle TRUE, \neg at(\mathtt{B}, v)\rangle$
		$\langle (in(u) \wedge (u \neq \mathtt{B})), at(u, w)\rangle$
		$\langle (in(u) \wedge (u \neq \mathtt{B})), \neg at(u, v)\rangle$

Since `carry-path` tasks have the same effects as `move-to` tasks, so do their associated ADL operators. Actually, `carry-path` can be modeled with the exact same operator; we do not need to specify any preconditions in the modeling ADL operator, but including necessary preconditions in the operator enables reasoning about an abstract task's preconditions prior to decomposing that task. For instance, suppose that we model `carry-path` with the operator

$$\boxed{\begin{array}{lll} \texttt{carry-path}(v,w) & \textsf{pre}: & (v \neq w) \wedge at(\texttt{B}, v) \\ & \textsf{eff}: & \langle TRUE, at(\texttt{B}, w)\rangle, \langle (in(u) \wedge (u \neq \texttt{B})), at(u,w)\rangle \end{array}}.$$

In this case a planner can reason about the initial location of the briefcase, and getting the appropriate contents into the case prior to expanding the subtasks.

The only restriction on a task τ_n's modeling ADL operator's preconditions is that they must be necessary after expanding τ_n's subtasks. Including extra, possibly unnecessary, preconditions can lead a planner to adding tasks that ultimately do not contribute to attaining a problem's goals; preconditions that are merely possible get addressed after the subtasks are expanded. For this reason, modeling tasks with preconditionless ADL operators is always safe, but adding necessary preconditions enables reasoning about them prior to decomposing the task.

Definition 11 *An* ADL *action a modeling an abstract task τ is* SOUND *if* DNF(Σ_ℓ^a) *always contains a singleton set Γ and $Q \in \Gamma$ guarantees that Q will be a required precondition of τ whenever ℓ is a required effect.*

Finally, we finish the development of our original example problem by discussing how to model `achieve[]` tasks; it turns out that these are much easier to model than `perform[]` tasks. Although we computed task `achieve[`ℓ`]`'s methods from the methods in $\mathcal{M}_p$, each method's E and P sets respectively denoted a single effect and no preconditions; thus the ADL operator for task `achieve[`ℓ`]` is

$$\boxed{\begin{array}{lll} \texttt{achieve[}\ell\texttt{]} & \textsf{pre}: & TRUE \\ & \textsf{eff}: & \langle TRUE, \ell\rangle \end{array}}$$

5.2 *Derivation Generation*

While UCPARSE uses precondition achievement to synthesize plans and derivation construction to determine their inclusion in the search space, UCREDUCE synthesizes plans by constructing derivations. As an adaptation of our revised HTN planning algorithm, UCREDUCE starts and progresses with the same five steps; UCREDUCE differs from the from the revised algorithm by using open goals to determine which subtasks to add and adding those subtasks during interaction resolution instead of the reduction step. Although figure 5.4 fully defines UCREDUCE's last four steps, we delay our discussion of the criticism loop until section 5.3.

Given our formal definition of UCREDUCE and the $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$ FHTN problem characterization, UCREDUCE has the initial invocation

$$\begin{aligned} \textsc{ucreduce}(&\langle T = \{\tau_0, \tau_\infty\}, B^{\emptyset}_{\mathtt{s}}, O^{\emptyset}_{\mathtt{s}}, L = \emptyset \rangle \\ &\quad \langle T_* = \{[\tau_{\mathtt{s}} \Rightarrow \langle \emptyset : T_{\mathtt{s}}, B_{\mathtt{s}}, O_{\mathtt{s}}, L_{\mathtt{s}} : \emptyset \rangle]\}, S_* = \emptyset, D_* = \emptyset \rangle, \\ &\quad \mathit{agenda} = \{\langle \rho @ \infty \rangle\}), \end{aligned}$$

where T_* contains the distinguished method for decomposing $\tau_{\mathtt{s}}$ with $\{\tau_0, \tau_\infty\} \subseteq T_{\mathtt{s}}$. This invocation reflects the requirement that a solution be reached by decomposing $\tau_{\mathtt{s}}$ to frugally attain the goal ρ at the end of the plan – denoted by the `stop` task τ_∞.

As before, we explain UCREDUCE by describing its behavior when solving the briefcase problem. While the initial tasks and auxiliary constraints are always the same, all other parameters are problem specific. In this case, *agenda* requires that the paycheck and briefcase respectively end up at home and the office; the initial derivation requires that the paycheck is at home because it stays at home while a `move-to(office)` task transports the briefcase.

Unlike UCPARSE, UCREDUCE succeeds in finding the solution in just three recursive calls, one for each nonprimitive task in the solution's corresponding derivation tree. Instead of implying that UCREDUCE is more efficient than UCPARSE, this observation implies that UCREDUCE performs more work during each call.

Algorithm UCREDUCE($\langle T, B, O, L\rangle, \langle T_*, S_*, D_*\rangle, agenda$)

1. **Criticism:** While *agenda* has selectable open conditions do:

   (a) **Open condition resolution:** Select $\langle Q_k@n_k\rangle$ (not referring to a member of D_*) in *agenda*, and resolve it with one of the two cases:

      i. **Non-literal:** If Q_k is not a literal, nondeterministically choose a conjunct Γ from the set DNF$(Q_k, \mathcal{I})$, remove $\langle Q_k@n_k\rangle$ and add $\langle \ell@n_k\rangle$ to *agenda* for each literal $\ell \in \Gamma$, and add Γ's constraints to B– failing on inconsistent B;

      ii. **Literal:** If Q_k is a literal, either fail if both $m \xrightarrow{\neg Q_k} n_k \in L$ and τ_{n_k} is primitive (an impossible plan) or SUPPORT $\langle Q_k@n_k\rangle$ with the steps in figure 5.9.

   (b) **Causal link protection:** Primitive task τ_l *threatens* a causal link $n \xrightarrow{\ell} m$ when it asserts $\neg\ell$ between primitives τ_n and τ_m. For each link $n \xrightarrow{\ell} m \in L'$ and each (l:`do`[a]) that might threaten it, nondeterministically choose to:

      i. **Promote:** Add $(m \prec l)$ to O, and fail if inconsistent.

      ii. **Demote:** Add $(l \prec n)$ to O, and fail if inconsistent.

      iii. **Confront:** Add $(n \prec l)$ and $(l \prec m)$ to O, and $\langle \neg\Sigma^a_{\neg\ell}@n\rangle$ to *agenda*.

2. **Termination:** If T has no abstract tasks, fail if $D_* \neq \emptyset$; otherwise succeed and return `sol`($\langle T, B, O, L\rangle$).

3. **Task selection:** Select and remove abstract task τ_n from T.

4. **Reduction:** Nondeterministically choose/add a new (instantiated from $\mathcal{M}$) method $[\tau_n \Rightarrow ctn_n]$ to T_*, add B_n to B, and add $O^{\emptyset}_n$ to O. Fail if B or O become inconsistent; otherwise remove all elements from L and D_* that mention τ_n. Respectively add $\langle \ell@n_i\rangle$ to *agenda* or $\langle n_i|\ell\rangle$ to D_* for each $n \xrightarrow{\ell} n_i$ or $n_i \xrightarrow{\ell} n$ removed.

5. **Recursive call:** Call UCREDUCE($\langle T, B, O, L\rangle, \langle T_*, S_*, D_*\rangle, agenda$)

Figure 5.4: The UCREDUCE algorithm for solving FHTN problems using reduction.

5.2.1 Agenda Maintenance

Upon initially calling UCREDUCE control starts with interaction resolution. Commonly, critic functions just add constraints between steps to disable unwanted interactions while facilitating desired ones, but UCREDUCE makes an extra requirement; the critic must also add tasks as specified by the derivation and required by the open goals in *agenda*. This addition devolves from an observation that resolving interactions between ADL actions might add new open goals to disable an effect. Since these open goals point to new tasks to add during interaction resolution, UCREDUCE might as well handle all open goals at this time. With respect to our example problem, the initial derivation and the goal $\langle(at(\texttt{B},\texttt{office}) \wedge at(\texttt{P},\texttt{home})@\infty\rangle$ drive the criticism loop to produce the task network appearing in figure 5.5.

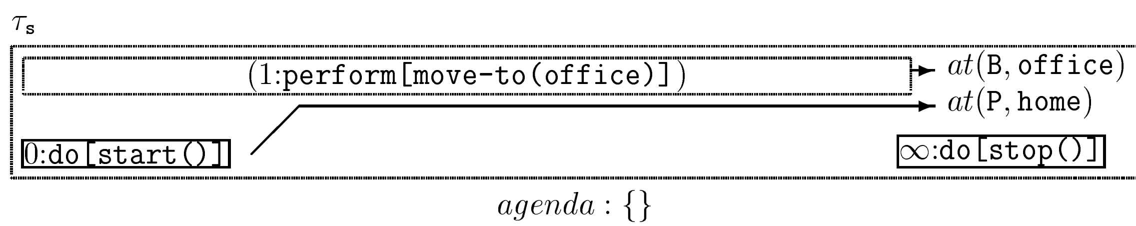


Figure 5.5: The task network after initial criticism

As figure 5.5 illustrates, task networks in a task reduction algorithm can contain non-primitive tasks; this has ramifications on how ordering constraints between tasks are handled. Essentially, a constraint $(i \prec j)$ implies that τ_i precedes τ_j, but the constraint becomes ambiguous if either task is nonprimitive. Should all subtasks of τ_i precede τ_j or just some? For this reason we disallow ordering constraints involving nonprimitive tasks, but we define distinguished moments $n.0$ and $n.\infty$ denoting when the nonprimitive task starts and ends respectively. Ordering constraints only involve primitive tasks and distinguished moments.

Due to our treatment of ordering constraints, adding a link $i \xrightarrow{\ell} j$ need not imply adding $(i \prec j)$; it just implies that some subtask of τ_i precedes a subtask of τ_j. Only when both tasks are primitive does adding $i \stackrel{\ell}{\rightarrow} j$ require adding $(i \prec j)$; thus although the problem requires that (1:`perform[move-to(office)]`) occurs between τ_0 and τ_∞, the added link $1 \stackrel{at(\mathtt{B},\mathtt{office})}{\rightarrow} \infty$ does not require any ordering constraint, but a constraint becomes required once the link is redirected to a primitive task.

Returning to UCREDUCE's behavior, control passes through termination to the tasks selection line because the computed task network contains a non-primitive task. Since the `move-to` task is the only non-primitive task, it is selected and removed. In general, there may be several tasks to choose from, and task selection deterministically chooses one; the choice is deterministic because the algorithm's completeness is not affected by the order in which the tasks are reduced.

5.2.2 Reduction

After selecting and removing a task τ_n control passes on to the reduction line, which decomposes τ_n with a nondeterministically chosen method. In our example, there is only one applicable decomposition method; thus this selection is forced. The line adds the new method to the derivation. Although there are no constraints to add in this case, reduction also involves adding the binding and ordering constraints to the task network.

Finally, line 4 strips out auxiliary constraints referring to the removed task; in our example this involves removing the link $1 \stackrel{at(\mathtt{B},\mathtt{office})}{\rightarrow} \infty$. In order to redirect the task's required effects, UCREDUCE adds the open goal $\langle at(\mathtt{B}, \mathtt{office})@\infty\rangle$ to *agenda*. By resolving this open goal UCREDUCE determines the actual subtasks of τ_1.

After updating *agenda* control passes to the recursive call and the second invocation of UCREDUCE begins. Once again control first visits the criticism line where tasks are added in accordance with the derivation and required by open goals in *agenda*; one of the resultant task networks appears in figure 5.6.

Although we delay discussing the criticism loop, it is worth mentioning how the

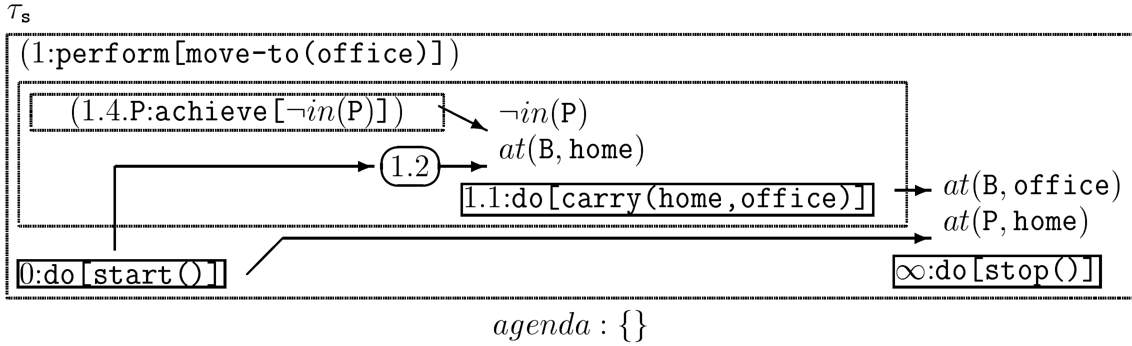


Figure 5.6: Derivation expanding (1:`perform[move-to(office)]`)'s subtasks

subtasks of the `move-to` task depend on context. The `carry` subtask always appears a `move-to` task's decomposition, but all other tasks depend on `carry`'s preconditions; in this case there were two. While the task always has an $at(\mathtt{B}, l)$ precondition, the $\neg in(\mathtt{P})$ precondition is required to assure that the paycheck stays at home. Since the second precondition arises from an interaction between the `carry` and the causal link requiring that the paycheck stay at home, determining whether or not to include the `achieve[¬in(P)]` subtask depends on context.

Once criticism handles all open goals to produce figure 5.6's task network, control passes through termination because the task network has a non-primitive task. Once again, task selection is forced to choose the only non-primitive task, and control passes on to reduction where the nondeterministic choice has two options to either phantomize the task or reduce it using a method computed from the `unload` task's definition. For now suppose that UCREDUCE chooses reduction; we discuss the phantomization case later. Using the reduction method, the reduction and the recursive call lines exhibit similar behavior as before, and the third invocation of UCREDUCE begins.

5.2.3 Termination

In the third call, the new derivation and *agenda*'s goal $\langle \neg in(\mathtt{P})@1.1\rangle$ combine to make the criticism loop extend the task network into the one illustrated in figure 5.7. Since all tasks are primitive, control now stops at the termination line, and UCREDUCE tests to see if the plan $\langle T, B, O, L\rangle$ is valid; this involves testing the soundness of the task network and derivation. Just like in UCPARSE a task network and its derivation are sound if all the network's tasks are accounted for in the derivation.

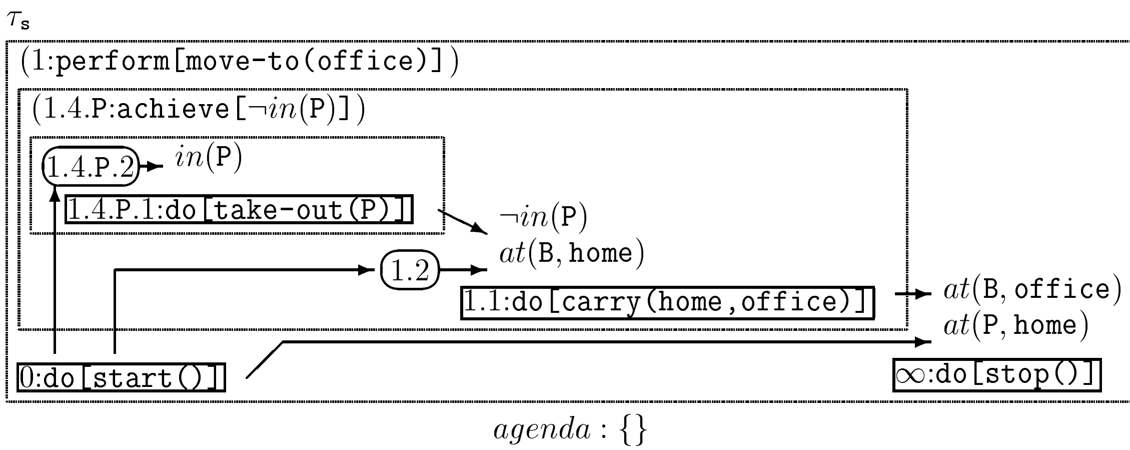


Figure 5.7: Derivation of the solution task network.

One case where the termination line would fail upon detecting a primitive task network arises if the `achieve` task was phantomized during the third call. If phantomization does occur, the criticism loop will extend the task network into the one illustrate it figure 5.8, where the point protection is supported by an as yet to be accounted for primitive task τ_n. Essentially, both UCREDUCE and UCPARSE take similar approaches toward handling point protections; they both add a primitive task to attain the point protection, and delay determining where the primitive belongs with respect to the problem's coordinated set of tasks. For this reason phantomizing the `achieve` task leads to failure because the resultant primitive task network has a task

τ_n that is not part of the problem's tasks.

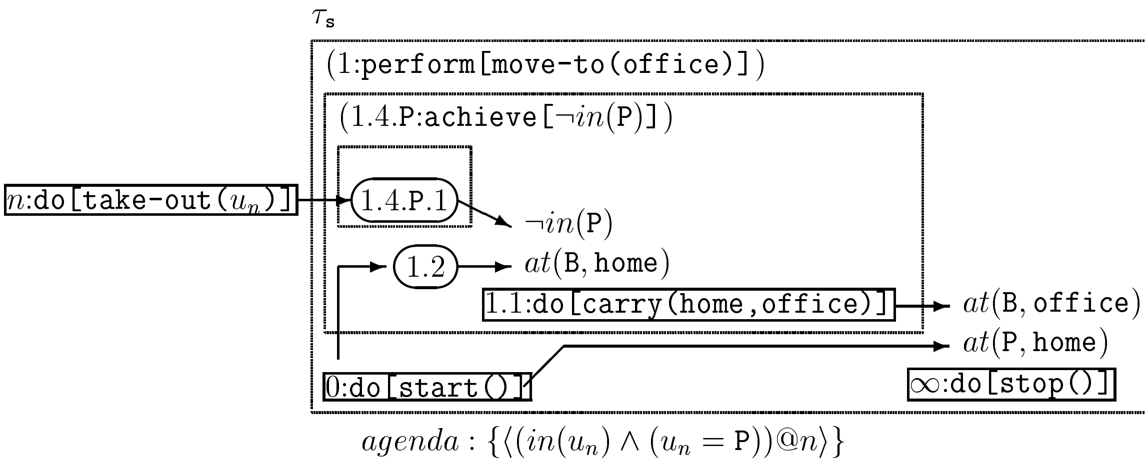


Figure 5.8: Derivation where (1.4.P:achieve[$\neg in$(P)]) was phantomized.

5.3 *Frugal Expansion*

Now that we understand the global behavior of the UCREDUCE algorithm, focusing on local behavior during the criticism loop becomes worthwhile. As illustrated back in figure 5.4, criticism is driven by a while loop that iterates until running out of selectable open goals, where an open goal is selectable if the derivation can determine which task provides the required effect. The purpose of the loop is to identify and instantiate as many of a task's subtasks as possible.

During each iteration of the loop, UCREDUCE removes an open condition from *agenda* and satisfies it by: (1) tracing the derivation's auxiliary constraints to determine the task τ responsible for attaining the open condition's literal; (2) adding a causal link to preserve the literal until needed; (3) adding open conditions needed by τ to produce the desired effect; and (4) protecting causal links. While the fourth step already appeared, the other three appear in figure 5.9.

Close inspection of the criticism loop and this figure shows that there are many

To SUPPORT a literal open condition $\langle Q_k@n_k\rangle$ do the following while backtracking whenever constraints B or O become inconsistent:

1. **Derivation traversal:** While ascending and descending, let $e \overset{b}{\in} S$ denote that there exists $e' \in S$ and $e \overset{B}{=} e' \backslash b$, where b denotes a substitution for universal variables.
   (a) **Ascend:** Starting with $n = n_k$, repetitively let $n = a(n)$ while $(\xrightarrow{Q_k} n) \overset{b}{\in} P_{a(n)}$ in T_*. Ultimately let $n = m$, where $(m \xrightarrow{Q_k} n) \overset{b}{\in} L_{a(n)}$ in T_*.
   (b) **Descend:** Repetitively let $n = m$ while $(m \xrightarrow{Q_k}) \overset{b}{\in} E_n$ in T_*.
2. **Universal variable resolution:** If n contains variables and $(T \cup S_*)$ lacks τ_n and (n', n), take all $[\tau_m \Rightarrow ctn_m]$, τ_m, or (m', m) from $(T_* \cup T \cup S_*)$, where $m \overset{b}{\cong} n$, and nondeterministically choose one or none. For those cases where one was chosen, add b to B and resume descend; Otherwise, add $O^n_{a(n)}$ to O and for each element add $m \ncong n$ to B.
3. **Point protection:** Handle point protections by:
   (a) if $(m, n) \in S_*$, then let $n = m$;
   (b) if n is a new point protection, nondeterministically choose $\tau_m = (m{:}\mathtt{do[}a\mathtt{]})$ from a set of existing (from T) or new (instantiated from $\mathcal{O}$) tasks that can assert Q_k, add (m, n) to S_*, add τ_m to D_* if τ_m is new, and finally let $n = m$;
   (c) if n is a task label denoting a new task $(n{:}\mathtt{do[}a\mathtt{]})$, nondeterministically choose $(n'{:}\mathtt{do[}a'\mathtt{]}) \in D_* \cup \{\tau_n\}$ such that $a \overset{b}{\cong} a'$. Let $B = B \cup b$, remove $(n'{:}\mathtt{do[}a'\mathtt{]})$ from D_*, and substitute n for n' in T, O, L, S_*, D_*, and *agenda*.
4. **Satisfaction:** Add $(n \xrightarrow{Q_k} n_k)$ to L, add τ_n to T if new, and add $(n \prec n_k)$ to O when both τ_n and τ_{n_k} are primitive (fail if O becomes inconsistent).
5. **Subgoal generation:** Where τ_n executes a, alter *agenda* to remove $\langle Q_k@n_k\rangle$, add $\langle pre(a)@n\rangle$ when τ_n new, and add $\langle \Sigma^a_{Q_k}@n\rangle$ when $\langle n|Q_k\rangle \notin D_*$ – else remove $\langle n|Q_k\rangle$.

Figure 5.9: Satisfying an open condition in accordance with derivation $\langle T_*, S_*, D_*\rangle$.

similarities between UCREDUCE and UCPARSE. The main difference involves UCPARSE's looping in its reduction step to decompose all abstract tasks before resolving the next open condition; UCREDUCE loops in the critic step to resolve all open conditions before reducing the next abstract task. Both algorithms trace a derivation's auxiliary constraints to determine the task to support an open condition.

Another difference between this UCREDUCE and UCPARSE involves the explicit inclusion of abstract tasks in a task network; UCPARSE only included primitives. Where UCPARSE adds an ordering constraint between two tasks for each added causal link, UCREDUCE will only add the constraint if both tasks are primitives. Also, a causal link between two tasks can only be threatened by a third if all three tasks are primitive.

5.3.1 Initial Task Network Expansion

Just like previous algorithm explanations, we discuss the criticism loop by describing its behavior when solving our briefcase problem. Upon initially invoking UCREDUCE, the task network, derivation, and *agenda* are as illustrated in figure 5.10. Since control starts with criticism, the initial responsibility of the loop is to identify which of $\tau_{\texttt{s}}$'s subtasks actually contribute toward attaining the problem's goals; this is done by resolving required preconditions as they arise.

$\tau_{\texttt{s}}$

$\boxed{\texttt{0:do[start()]}}$ $\boxed{\infty\texttt{:do[stop()]}}$

$$agenda : \{\langle (at(\texttt{B}, \texttt{office}) \wedge at(\texttt{P}, \texttt{home}))@\infty \rangle\}$$

Figure 5.10: Derivation of the initial null task network.

Upon entering the loop control passes to open condition resolution; where the single condition in the agenda is selected. After removing this condition $\langle Q_k @ n_k \rangle$, UCREDUCE discovers that Q_k is a conjunctive formula. Just like UCPARSE, UCREDUCE calls $\textsc{dnf}(Q_k, \mathcal{I})$ to break Q_k into literals by computing its disjunctive normal form; as before, Γ contains the literals $at(\texttt{B}, \texttt{office})$ and $at(\texttt{P}, \texttt{home}))$. After adding these literals' associated open conditions to the *agenda*, control runs around the loop to revisit the open condition resolution line, which now selects goal $\langle at(\texttt{P}, \texttt{home}))@\infty\rangle$. Since Q_k is now a literal, the literal condition case of open condition resolution now applies, and UCREDUCE tests for an impossible plan.

Testing for an impossible plan changes when tasks can be abstract. A plan is impossible if it requires both Q_k and $\neg Q_k$ at the same time. When a task is primitive, it is atomic and cannot have both Q_k and $\neg Q_k$ as preconditions; such is not the case for abstract tasks which can have temporal extent. Thus the impossibility test only applies when the chosen open condition's task is primitive. In this case the test applies, but does not cause failure.

After the impossible plan test, UCREDUCE uses derivation traversal to determine that the initial conditions, or τ_0, are responsible for supporting $\langle at(\texttt{P}, \texttt{home}))@\infty\rangle$. This traversal starts with n equal to ∞ and immediately finds $0 \overset{at(\texttt{P},\texttt{home}))}{\rightarrow} \infty$ in $L_{\texttt{s}}$. Since τ_0 is an existing primitive task in T, control passes successively through the satisfaction and subgoal generation steps, which respectively add $0 \overset{at(\texttt{P},\texttt{home}))}{\rightarrow} \infty$ to L and $\langle TRUE@0\rangle$ to *agenda* before control runs to the third iteration of the critic loop.

Just like the seventh call of UCPARSE, the third iteration of the critic loop selects and resolves the non-literal goal $\langle TRUE@0\rangle$ by having DNF return $\{\{\}\}$. At this point the non-literal subcase of open condition resolution adds nothing to B or *agenda* and control passes on to the fourth iteration of the critic loop with the derivation illustrated in figure 5.11.

On the fourth iteration through the critic loop, the open condition resolution line removes *agenda*'s last goal, requiring the briefcase at the office. Since τ_∞ is a subtask of $\tau_{\texttt{s}}$ derivation traversal immediately finds $1 \overset{at(\texttt{B},\texttt{office})}{\rightarrow} \infty$ in $L_{\texttt{s}}$, where τ_1 performs a `move-to`. Since the label 1 contains no variables and τ_1 is a new non-primitive task,

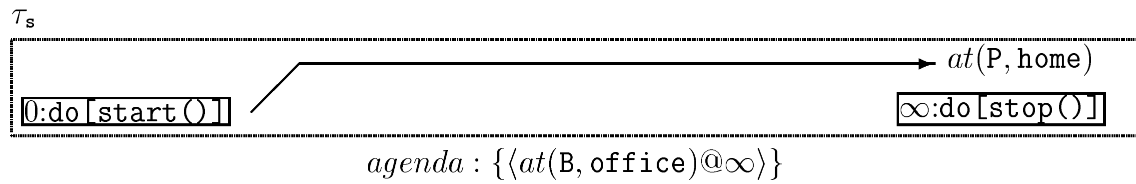


Figure 5.11: Derivation after satisfying $\langle at(\mathtt{P}, \mathtt{home})@\infty\rangle$.

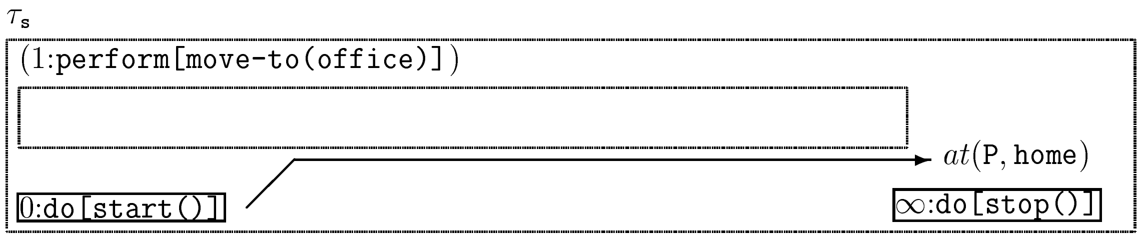
control passes to the satisfaction and subgoal generation step which adds the link $1 \overset{at(\mathtt{B},\mathtt{office})}{\rightarrow} \infty$ to L.

At this point the subgoal generation line refers to table 5.1 to determine the goal $\langle TRUE \wedge TRUE@1\rangle$ to add to *agenda*. Just as in the third iteration of the critic loop, the new goal evaporates during fifth iteration because DNF returns $\{\{\}\}$. Upon evaporating the goal, the *agenda* becomes empty and control passes out of the critic loop with the task network after initial criticism, back in figure 5.5.

5.3.2 Subtask Expansion

Where the first critic loop's invocation involved expanding the problems initial tasks as required by its goals, the second execution involves expanding the subtasks of $\tau_1 = (1$:`perform[move-to(office)]`$)$ in accordance with its context within the plan. Immediately prior to the second invocation, UCREDUCE strips τ_1 out of the task network and adds a decomposition method to the derivation. While stripping τ_1 out, the preconditions it supports are reintroduced as open goals; the resultant task network with its derivation appears in figure 5.12, where the box to ultimately contain τ_1's subtasks is empty until the critic loop identifies the appropriate subtasks.

On the first pass through the critic loop, open condition resolution involves selecting the lone goal in *agenda*, making the assignment $n = \infty$, and passing control on to the literal subcase. Since $L_{\mathtt{s}}$ contains $1 \overset{at(\mathtt{B},\mathtt{office})}{\rightarrow} \infty$, derivation traversal's ascend line assigns 1 to n and passes control to the descend line, where $(1.1 \overset{at(\mathtt{u},\mathtt{office})}{\rightarrow})$ is found



$agenda : \{\langle at(\texttt{B}, \texttt{office})@\infty\rangle\}$

Figure 5.12: Derivation after reducing (1:perform[move-to(office)]).

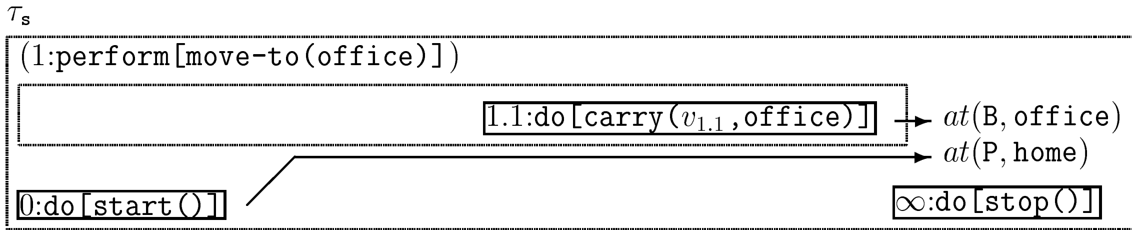
in E_1; thus derivation traversal involves tracing the auxiliary constraints to find task $\tau_{1.1}$. Due to 1.1's lacking variables and D_* being empty, control passes uneventfully through to the satisfaction line, which adds $\tau_{1.1}$ to T and assures that $\tau_{1.1}$'s effect persists until the end of the plan by adding $1.1 \overset{at(\texttt{B},\texttt{office})}{\rightarrow} \infty$ since both $\tau_{1.1}$ and τ_∞ are primitive, the satisfaction line also adds the constraint $(1.1 \prec \infty)$.

In order to execute task $\tau_{1.1}$ to carry the briefcase to the office, the briefcase must already be at some location $v_{1.1}$ other than the office; to assure that this precondition gets satisfied, the subgoal generation line adds a goal to *agenda*. Just as in UCPARSE's third call, the computed subgoal is

$$\langle (v_{1.1} \neq \texttt{office}) \wedge at(\texttt{B}, v_{1.1}) \wedge ((TRUE \wedge (\texttt{office} = \texttt{office})) \vee \\ (in(\texttt{B}) \wedge (\texttt{B} \neq \texttt{B}) \wedge (\texttt{office} = \texttt{office})))@1.1\rangle.$$

Control now passes on to detect a threat during causal link protection. Essentially, $\tau_{1.1}$ threatens $0 \overset{at(\texttt{P},\texttt{home})}{\rightarrow} \infty$ by moving the paycheck. Although UCREDUCE can make a nondeterministic choice over 3 options, only confrontation avoids making O inconsistent; thus the confront line addresses the threat by adding an open goal to *agenda*. This goal assures that the paycheck is outside the briefcase when it is moved.

At this point the first iteration of the loop ends with the resultant task network, and its *agenda*, appearing in figure 5.13.



$$\begin{aligned} agenda : \{ & \langle (v_{1.1} \neq \texttt{office}) \wedge at(\texttt{B}, v_{1.1}) \wedge \\ & \quad ((TRUE \wedge (\texttt{office} = \texttt{office})) \vee \\ & \quad (in(\texttt{B}) \wedge (\texttt{B} \neq \texttt{B}) \wedge (\texttt{office} = \texttt{office})))@1.1 \rangle, \\ & \langle (\neg in(\texttt{P}) \vee (\texttt{B} = \texttt{P}))@1.1 \rangle \} \end{aligned}$$

Figure 5.13: Derivation after adding (1.1:do[carry($v_{1.1}$,office)]).

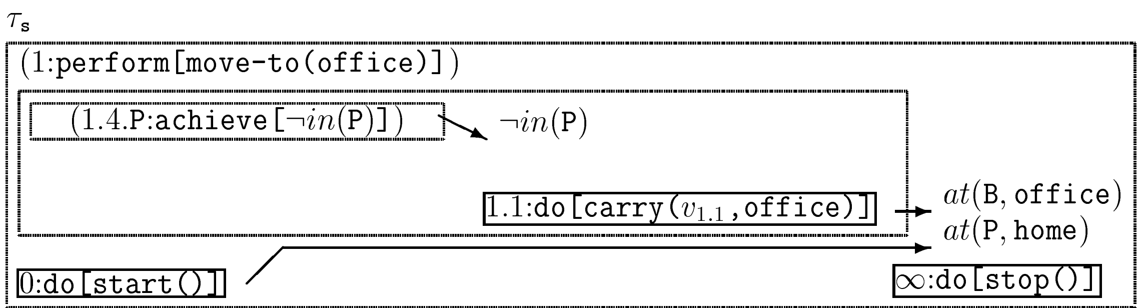
5.3.3 Universal Variables

On the second iteration of the loop, there are two goals in *agenda*, and open condition resolution involves selecting and removing one. Since the order of selection is arbitrary, suppose that UCREDUCE removed $\langle (\neg in(\texttt{P}) \vee (\texttt{B} = \texttt{P}))@1.1 \rangle$. Since this goal is non-literal, UCREDUCE breaks it up and adds $\langle \neg in(\texttt{P})@1.1 \rangle$ to *agenda* in order to reselect it on the third iteration of the critic loop, where universal variables first appear.

In the third iteration, after selecting to resolve $\langle \neg in(\texttt{P})@1.1 \rangle$, control starts derivation traversal with 1.1 assigned to n, and ascending immediately finds $1.4.\mathbf{b} \xrightarrow{\neg in(\mathbf{b})} 1.1$, where $\mathbf{b}$ is a universal variable. Substituting P for $\mathbf{b}$ produces the label 1.4.P, and the descent starts with 1.4.P assigned to n. Although n denotes a label affected by universal quantification, control uneventfully passes through the universal variable resolution line because there are no tasks with labels that might match 1.4.P; control also passes through the point protection line because the label denotes a task.

In general, the possibility of having a goal like $\langle \neg in(x)@1.1 \rangle$, leads to tasks with variablized labels. If our task network had a task like (1.4.x:achieve[$\neg in(x)$]), line universal variable would have to consider adding the variable constraint (x = P) and jumping back to the descent part of derivation traversal..

Returning to our trace, the satisfaction line adds task $\tau_{1.4.\mathtt{P}}$ and $1.4.\mathtt{P} \overset{\neg in(\mathtt{P})}{\rightarrow} 1.1$; since a task was added, subgoal generation adds $\langle TRUE \wedge TRUE@1.4.\mathtt{P}\rangle$ to *agenda*, and the fourth iteration evaporates this goal. leaving the derivation and agenda found in figure 5.14.



$$\begin{aligned} agenda : \{\langle(v_{1.1} \neq \mathtt{office}) \wedge at(\mathtt{B}, v_{1.1}) \wedge \\ ((TRUE \wedge (\mathtt{office} = \mathtt{office})) \vee \\ (in(\mathtt{B}) \wedge (\mathtt{B} \neq \mathtt{B}) \wedge (\mathtt{office} = \mathtt{office})))@1.1\rangle\} \end{aligned}$$

Figure 5.14: Derivation after adding (1.4.P:achieve[$\neg in(\mathtt{P})$]).

5.3.4 Point Protection Resolution

While the critic loop's fifth iteration breaks up *agenda*'s non-literal goal and replaces it with $\langle at(\mathtt{B}, v_{1.1})@1.1\rangle$, its sixth iteration resolves the replacement goal by using a point protection. The goal and the auxiliary constraint $[1.2] \overset{at(\mathtt{B},v_{1.1})}{\rightarrow} 1.1$ in L_1 imply that derivation traversal ultimately assigns 1.2 to n. Since 1.2 has no variables, control passes through universal variable resolution resulting in figure 5.15's derivation.

Since 1.2 denotes a new point protection, the second case of the point protection step directs UCREDUCE to either add a new primitive task or reuse an existing one to support the point protection. Since primitive task τ_0 can satisfy the precondition, suppose that line UCREDUCE chooses it and makes n denote τ_0 by adding $(1.2, 0)$ to the symbol table S_*; D_* remains unchanged since τ_0 already existed, and control passes through the satisfaction and subgoal generation step which respectively add

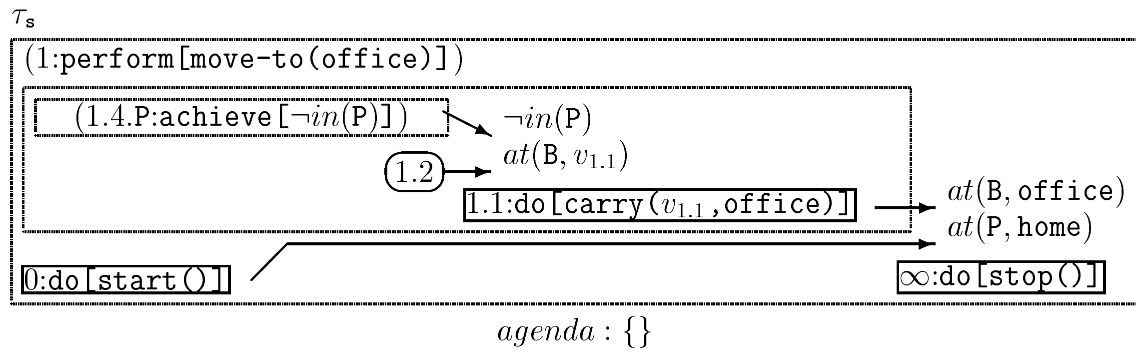


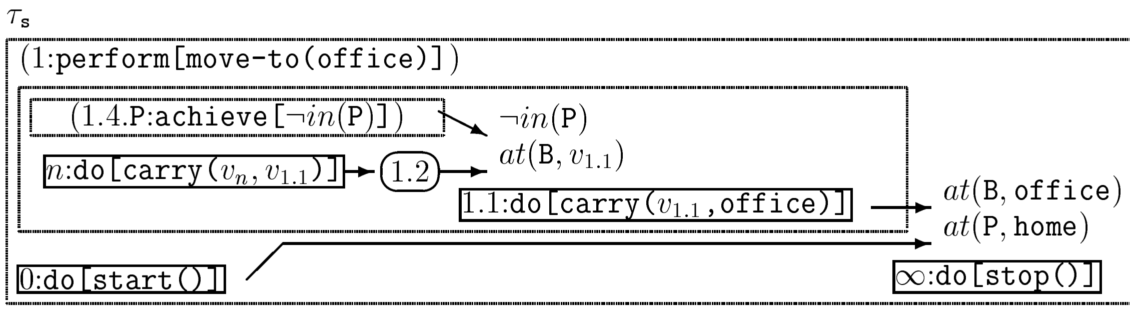
Figure 5.15: Derivation after finding point protection.

$0 \overset{at(\mathtt{B},v_{1.1})}{\rightarrow} 1.1$ to L and $\langle(TRUE \wedge (v_{1.1} = \mathtt{home})@0\rangle$ to *agenda*. Also, τ_0 and $\tau_{1.1}$ are both primitive, and UCREDUCE adds a $(0 \prec 1.1)$ constraint to O.

At this point the critic loop's seventh iteration starts and the sixth iteration's added goal is resolved in the non-literal subcase of open condition resolution. The only result of this iteration involves adding $(v_{1.1} = \mathtt{home})$ to B.

Although the previous trace shows how interaction resolution produces the expected task network, there is another solution; suppose that instead of using the initial conditions to satisfy point protection $1.2 \overset{at(\mathtt{B},v_{1.1})}{\rightarrow} 1.1$, the point protection line added a primitive task $\tau_n = (n{:}\mathtt{do[carry(}v_n, w_n\mathtt{)]})$. Since this task has yet to be identified within the problem's coordinated tasks, it gets added to D_*, the delayed tasks. UCREDUCE continues and updates the task network and *agenda* during satisfaction and subgoal generation. Finally, the critic loop detects a threat to the requirement that the paycheck stay at home and resolves it in the same manner as the threat from $\tau_{1.1}$. When the critic loop completes, the task network appears in figure 5.16.

Since the second goal in *agenda* is not a literal, the loop makes an abbreviated forth iteration, where line 1.a breaks down the second goal, adds constraints to B, and adds goal $\langle at(\mathtt{B}, v_n)@n\rangle$ to *agenda*. Upon returning to line 1, UCREDUCE notices



$$agenda : \{\langle(v_n \neq v_{1.1}) \wedge at(\mathtt{B}, v_n) \wedge ((TRUE \wedge (v_{1.1} = v_{1.1}))\vee (in(\mathtt{B}) \wedge (\mathtt{B} \neq \mathtt{B}) \wedge (v_{1.1} = v_{1.1})))@1.1\rangle \; \langle(\neg in(\mathtt{P}) \vee (\mathtt{P} = \mathtt{B}))@n\rangle\}$$

Figure 5.16: Derivation after finding point protection.

that none of *agenda*'s goals are selectable, and interaction resolution completes with two open goals and a task in D_*.

At this point we see that interaction resolution can produce several task networks when expanding a task's subtasks; there were two that conformed with the expanded `move-to` task. While the first leads to a solution, the second ultimately leads to a dead end because there is no way to further expand the derivation to account for task τ_n. Although there are several other points during planning where UCREDUCE executes interaction resolution, we stop our discussion here because subsequent executions follow the same reasoning as the two executions previously explained.

5.4 Summary

In this chapter we presented UCREDUCE, an algorithm based on reduction that solves FHTN problems. There are two differences between UCREDUCE and standard reduction-based hierarchical task-network planners. The first involves modifying the termination test, reduction, then interaction resolution loop into an interaction resolution, termination test, then reduction loop. The second involves expanding interaction resolution to also frugally determine the subtasks of reduced tasks. In order to frugally expand tasks with non-primitive subtasks, we also showed how to model non-primitive tasks with ADL actions.

To illustrate how UCREDUCE generates its search space, we traced its behavior while solving our example briefcase problem. Although UCREDUCE only took 3 recursive calls instead of UCPARSE's 13 recursive calls, a striking similarity between the two algorithms becomes apparent when we also note the results of each criticism loop iteration. The trace is summarized with the following items.

1. Reduce (1:`perform[move-to(office)]`) after iterating five times through the criticism loop to successively:

   (a) break $\langle (at(\texttt{P}, \texttt{home}) \wedge at(\texttt{B}, \texttt{office}))@\infty \rangle$ up into its components;

   (b) use initial conditions to support $\langle at(\texttt{P}, \texttt{home})@\infty \rangle$;

   (c) resolve $\langle TRUE@0 \rangle$, which evaporates;

   (d) add (1:`perform[move-to(office)]`) to support $\langle at(\texttt{B}, \texttt{office})@\infty \rangle$; then

   (e) resolve $\langle TRUE \wedge TRUE@1 \rangle$, which evaporates;

2. Reduce (1.4.P:$\texttt{achieve[}\neg in(\texttt{P})\texttt{]}$) after iterating seven times through the criticism loop to successively:

   (a) add (1.1:$\texttt{do[carry(}v_{1.1}\texttt{, office)]}$) to support $\langle at(\texttt{B}, \texttt{office})@\infty \rangle$;

   (b) break $\langle (\neg in(\texttt{P}) \vee (\texttt{P} = \texttt{B}))@1.1 \rangle$ up into its components;

   (c) add (1.4.P:$\texttt{achieve[}\neg in(\texttt{P})\texttt{]}$) in order to support $\langle \neg in(\texttt{P})@1.1 \rangle$;

   (d) resolve $\langle TRUE \wedge TRUE@1.4.\texttt{P} \rangle$, which evaporates;

   (e) break $\langle (v_{1.1} \neq \texttt{office}) \wedge at(\texttt{B}, v_{1.1}) \wedge ((TRUE \wedge (\texttt{office} = \texttt{office})) \vee (in(\texttt{B}) \wedge (\texttt{B} \neq \texttt{B}) \wedge (\texttt{office} = \texttt{office})))@1.1 \rangle$ up into its components;

   (f) use initial conditions to support $\langle at(\texttt{B}, v_{1.1})@1.1 \rangle$; then

(g) resolve $\langle TRUE \wedge (v_{1.1} = \texttt{home})@0\rangle$ by adding the binding constraint;

3. Return solution plan after iterating four times through the criticism loop to successively:

   (a) add (1.4.P.1:`do[take-out(P)]`) to support $\langle\neg in(\texttt{P})@1.1\rangle$;

   (b) break $\langle in(\texttt{P}) \wedge TRUE \wedge (\texttt{P} = \texttt{P})@1.4.\texttt{P}.1\rangle$ up into its components;

   (c) use initial conditions to support $\langle in(\texttt{P})@1.4.\texttt{P}.1\rangle$; then

   (d) resolve $\langle TRUE@0\rangle$, which evaporates.

Chapter 6

THEORETICAL CONSIDERATIONS

In the previous three chapters we successively presented FHTN planning problems, how to extend a PO planner to solve an FHTN problem, and how to similarly extend an HTN planner. In this chapter we prove that both planners are sound and complete. Our proofs take the FHTN problem definition from chapter 3 and show that the algorithms from chapters 4 and 5 are guaranteed to only return solutions, and for each possible solution there exists appropriate sets of nondeterministic choices to guide UCPARSE and UCREDUCE to the solution.

Instead of proving soundness and completeness for each algorithm separately, we can save some effort by taking advantage of the identical calling conventions of both algorithms; given an FHTN problem $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$, the initial invocations are

$$\begin{aligned}
\text{UCPARSE}(\quad & \langle T = \{\tau_0, \tau_\infty\}, B^{\emptyset}_{\mathsf{s}}, O^{\emptyset}_{\mathsf{s}}, L = \emptyset \rangle \\
& \langle T_* = \{[\tau_{\mathsf{s}} \Rightarrow \langle \emptyset : T_{\mathsf{s}}, B_{\mathsf{s}}, O_{\mathsf{s}}, L_{\mathsf{s}} : \emptyset \rangle]\}, S_* = \emptyset, D_* = \emptyset \rangle, \\
& \mathit{agenda} = \{\langle \rho @ \infty \rangle\}) \text{ and} \\
\text{UCREDUCE}(& \langle T = \{\tau_0, \tau_\infty\}, B^{\emptyset}_{\mathsf{s}}, O^{\emptyset}_{\mathsf{s}}, L = \emptyset \rangle \\
& \langle T_* = \{[\tau_{\mathsf{s}} \Rightarrow \langle \emptyset : T_{\mathsf{s}}, B_{\mathsf{s}}, O_{\mathsf{s}}, L_{\mathsf{s}} : \emptyset \rangle]\}, S_* = \emptyset, D_* = \emptyset \rangle, \\
& \mathit{agenda} = \{\langle \rho @ \infty \rangle\}),
\end{aligned}$$

where T_*, contains the distinguished method for decomposing τ_{s} with $\{\tau_0, \tau_\infty\} \subseteq T_{\mathsf{s}}$. Not only are the initial invocations identical, but both algorithms use the parameters the same way in recursive calls.

In many respects each recursive call denotes a partial plan and our algorithms search through a space of partial plans by making nondeterministic choices to compute the next recursive call. In this chapter we prove invariants on these partial plans and use these invariants to prove soundness and completeness. For convenience, we will use the term *partial plan* to denote a task network $\langle T, B, O, L\rangle$, derivation $\langle T_*, S_*, D_*\rangle$, and *agenda* at a given point in the planning process.

6.1 Recursive Invariants

As mentioned back in section 3.3.2, an action sequence $\{a_i\}_{i=0}^{k}$ solves an FHTN planning problem $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho\rangle$ if and only if it is goal directed to attain ρ and task directed to perform a justified member of $\mathcal{L}(\mathcal{M})$. In our soundness and completeness proofs we start by proving three recursive invariants. While the first two are devoted toward proving soundness, the third assists in the completeness proof. Given these invariants we prove through induction that any action sequence returned by either UCPARSE or UCREDUCE solves the problem, and there exists appropriate nondeterministic choices to guide each algorithm to find a solution if one exists.

6.1.1 Attaining Goals

Our first recursive invariant involves frugally attaining ρ; it is partially defined in terms of $required_n$ – a Boolean equation of literals and variable constraints that must hold prior to executing primitive task τ_n's action "a". As equation 6.1 shows, $required_n$ has a component for the action's precondition, two components for those extra preconditions needed to provide desired effects, and a fourth component needed to avoid undesired effects. The need for two desired effect components stems from the observation that while all desired effects eventually reside in L, some temporarily appear in D_*.

$$required_n = pre(a) \wedge \left(\bigwedge_{n \xrightarrow{\ell} o \in L} \Sigma^a_\ell\right) \wedge \left(\bigwedge_{\langle n|\ell\rangle \in D_*} \Sigma^a_\ell\right) \wedge \left(\bigwedge_{\ell \in preserved_n} \Pi^a_\ell\right) \tag{6.1}$$

While the first three components are self contained, the fourth refers to equation 6.2, which defines contributions from other actions that τ_n must not affect.

$$preserved_n = \{\ell \mid m \xrightarrow{\ell} o \in L \text{ and } \{\tau_m \prec \tau_n, \tau_n \prec \tau_o\} \subseteq O\} \tag{6.2}$$

Given the required precondition equations for each primitive task, the first invariant specifies that *agenda* records those goals whose satisfaction, combined with properties already guaranteed, will assure each step's required precondition's satisfaction. Thus a threatless partial plan is goal oriented when the goals in *agenda* assure that the required preconditions specified in the plan will hold; we say that a plan is *threatless* when all primitive tasks that might affect a primitive link's condition are ordered with respect to the link.

Definition 12 *A partial plan with threatless $\langle T, B, O, L\rangle$ is* GOAL ORIENTED *when there exists a set of bindings $B_n \subseteq B$ for any primitive task $\tau_n \in T$ such that* $\text{DNF}(assured_n, \mathcal{I}) \subseteq \text{DNF}(required_n, \mathcal{I})$*, where*

$$assured_n = \left(\bigwedge_{\langle \rho @ n\rangle \in agenda} \rho\right) \wedge \left(\bigwedge_{m \xrightarrow{\ell} n \in L} \ell\right) \wedge \left(\bigwedge_{b \in B_n} b\right).$$

Notice that the equation defining $assured_n$ has three components, one from *agenda* and the others from the task network's causal links and variable bindings. Eventually *agenda* becomes empty and the causal links combined with the bindings fully support the required preconditions, but initially the task network has no causal links and we prove the invariant as follows.

Lemma 1 *The initial partial plan in calls to* UCPARSE *and* UCREDUCE *is goal oriented, where $B_n = \emptyset$ for each primitive task.*

Proof: Given that L and D_* have no links or delayed required effects in the initial calls, the equations for $required_n$ and $assured_n$ for task $(n{:}\texttt{do}[a])$ come respectively from $pre(a)$ and *agenda*. Since τ_0's action has no preconditions and *agenda* initially refers to only τ_∞, both $\textsc{dnf}(required_0, \mathcal{I})$ and $\textsc{dnf}(assured_0, \mathcal{I})$ are the set $\{\{\}\}$. Unlike τ_0, the other task τ_∞ has the precondition ρ; since *agenda* has the element $\langle \rho@\infty \rangle$, both $assured_\infty$ and $required_\infty$ denote the equation ρ, and $\textsc{dnf}(assured_\infty, \mathcal{I})$ denotes the same set as $\textsc{dnf}(required_\infty, \mathcal{I})$. Since T only has these two tasks, the partial plan is goal oriented. $\square$

6.1.2 Performing Tasks

Our second recursive invariant involves performing task τ_{s} as defined by $\mathcal{M}$. To record how a partial plan is derived from τ_{s} the three parameters in a derivation $\langle T_*, S_*, D_* \rangle$ respectively denote the methods for decomposing τ_{s}, which action satisfies which point protection, and those tasks and assured effects not currently matching features derived from τ_{s}. With this invariant we show how our planners use derivations to record how τ_{s} decomposes to produce the task network $\langle T, B, O, L \rangle \backslash \theta$ for any grounding substitution θ satisfying B.

Definition 13 *A partial plan with $\langle T, B, O, L \rangle$ and $\langle \{d_1 \ldots d_k\}, S_*, D_* \rangle$ is* TASK ORIENTED *iff every grounding substitution θ satisfying B has a set of methods $\{d_1^\theta \ldots d_k^\theta\}$ such that $d_i^\theta \in \mathcal{G}^*(\{d_i\})$ and*

$$\tau_{\mathsf{s}} \stackrel{d_1^\theta}{\Rightarrow} ctn_1 \stackrel{d_2^\theta}{\Rightarrow} \ldots \stackrel{d_k^\theta}{\Rightarrow} \langle \emptyset : T^\theta, B^\theta, O^\theta, L^\theta : \emptyset \rangle,$$

where 5 conditions hold: (1) $T\backslash\theta = T^\theta \cup \{\tau\backslash\theta \mid \tau \in D_*\}$; (2) B^θ holds; (3) $O^\theta \subseteq O\backslash\theta$; (4) $L\backslash\theta = \{n \xrightarrow{\ell} m \mid (\tau_n \in T^\theta$ and $n \xrightarrow{\ell} m \in L^\theta)$ or $((n, o) \in S_*\backslash\theta$ and $o \xrightarrow{\ell} m \in L^\theta)\}$; and (5) every $\tau_n \in T$ other than τ_0 and τ_∞ either appears in D_* or has at least one corresponding causal link $n \xrightarrow{p} m$ in L^θ.

As listed in the invariant's definition, there are five constraints; the first four match the components of the grounded task network $\langle T, B, O, L \rangle \backslash \theta$ against the components of a task network $\langle T^\theta, B^\theta, O^\theta, L^\theta \rangle$ derivable from τ_{s}, and the fifth is used for justifying tasks in T. In the first constraint we match elements of $T\backslash\theta$ with those of

T^θ; the D_* component of this matching catches those tasks in T that satisfy point protections and might eventually become derivable from τ_{s}. While the second and third constraints directly match binding and ordering constraints, the fourth constraint uses an indirection through S_* when matching point protections with causal links.

Eventually, as all tasks in T become primitive and both *agenda* and D_* become empty, this invariant will imply that the return action sequence is derivable from τ_{s}. Initially this invariant holds primarily because the first task network was computed from $\mathcal{M}$'s sole method for decomposing τ_{s}.

Lemma 2 *The initial calls to* UCPARSE *and* UCREDUCE *have task oriented partial plans.*

Proof: Recall the definition of $\mathcal{G}^*(\mathcal{M})$ using the equation

$$\begin{aligned}\mathcal{G}^*(\mathcal{M}) = \{\ [\tau_n \Rightarrow \langle P' : T', B_n, O_n, L' : E'\rangle]\ |\ \\ [\tau_n \Rightarrow \langle P_n : T_n, B_n, O_n, L_n : E_n\rangle] \in \mathcal{G}(\mathcal{U}(\mathcal{M} \cup \mathcal{A}(\mathcal{M}))), \\ P' \subseteq P_n, T' \subseteq T_n, L' \subseteq L_n, \text{ and } E' \subseteq E_n\},\end{aligned}$$

where $\mathcal{G}, \mathcal{U}$, and $\mathcal{A}$ are functions that take a set of methods and compute another set by respectively grounding variables, expanding universal variables, and determining achieve tasks corresponding to a perform task. Due to the subsets on T_n and L_n, this definition gives us

$$[\tau_{\mathsf{s}} \Rightarrow \langle \emptyset : T = \{\tau_0, \tau_\infty\}, B_{\mathsf{s}}, O_{\mathsf{s}}, L = \emptyset : \emptyset\rangle]\backslash\theta \in \mathcal{G}^*(\{[\tau_{\mathsf{s}} \Rightarrow \langle \emptyset : T_{\mathsf{s}}, B_{\mathsf{s}}, O_{\mathsf{s}}, L_{\mathsf{s}} : \emptyset\rangle]\})$$

for any θ satisfying B_{s}. Letting d^θ be this method we get

$$\tau_{\mathsf{s}} \overset{d^\theta}{\Rightarrow} \langle \emptyset : T^\theta, B^\theta, O^\theta, L^\theta : \emptyset\rangle,$$

where the 5 conditions hold: $T\backslash\theta = T^\theta$; $B^\theta = B_{\mathsf{s}}\backslash\theta$ holds; $O^\theta = O_{\mathsf{s}}\backslash\theta$; both L^θ and $L\backslash\theta$ are empty; and there are no tasks other than τ_0 and τ_∞. $\square$

6.1.3 Finding Solutions

While our first two recursive invariants are devoted toward proving soundness, our third helps prove completeness. By defining a partial plan's candidate set semantics [Kambhampati et al. 94], we can characterize the partial plan at any given moment as a set of zero or more solution sequences. Intuitively, this set consists of those sequences that our algorithms can produce from the current state of the arguments. We formally define this set in terms of solution sequences *consistent* with the current partial plan.

To define consistency we characterize a solution sequence's causal structure as a set of *causal supports* each of which denotes a set of causal links used to assure that an action a can execute, has a desired effect, or lacks a destructive side effect. During planning, identifying a new action a, desired effect Q, or undesired effect Q of a task τ_n respectively results in causing a causal support's creation by adding the initial open condition $\langle pre(a)@n\rangle$, $\langle\Sigma^a_Q@n\rangle$, or $\langle\Pi^a_Q@n\rangle$.

A partial plan's auxiliary constraints and *agenda* combine to form a set of causal constraints, where a *causal constraint* is either a singleton set with the initial open condition $\langle Q@n\rangle$, or a set of variable constraints, causal links, and literal open conditions supporting $\textsc{dnf}(Q)$ for τ_n. A solution's causal support matches a causal constraint when a grounding substitution can make both involve assuring the same set of conditions for the same step. We say that a causal constraint is *finalized* when it lacks open conditions; here the causal constraint and causal support are equivalent.

Definition 14 *A solution sequence $\{a_i\}_{i=0}^k$ and its decomposition*

$$\tau_{\mathsf{s}} \overset{d_1^{\mathsf{s}}}{\Rightarrow} ctn_1 \overset{d_2^{\mathsf{s}}}{\Rightarrow} \ldots \overset{d_k^{\mathsf{s}}}{\Rightarrow} \langle\emptyset : T^{\mathsf{s}}, B^{\mathsf{s}}, O^{\mathsf{s}}, L^{\mathsf{s}} : \emptyset\rangle,$$

are CONSISTENT *with a partial plan if there exists a grounding substitution θ^{s} satisfying B such that four conditions hold: (1) each method $d_i \in T_*$ has a $d_i^{\mathsf{s}} \in \mathcal{G}^*(\{d_i\})$, (2) each task $\tau_n \in T\backslash\theta^{\mathsf{s}}$ either appears in T^{s} or has a decomposing method d_i^{s}, (3) $O\backslash\theta^{\mathsf{s}}$*

is consistent with the mapping π, and (4) the causal constraints in $(L \cup agenda)\backslash\theta^{\mathsf{s}}$ match distinct causal supports.

By defining the solution sequences consistent with a partial plan, we characterize its candidate sets of sequences. To prove completeness, we hypothesize some arbitrary solution sequence $\{a_i\}_{i=0}^{k}$ and show that the algorithms can produce it by first proving its presence in the initial call's candidate set and then proving that each call can keep it in the candidate set given the appropriate nondeterministic choices.

Lemma 3 *Any solution sequence is consistent with both planners' initial partial plan.*

Proof: Consider the four conditions. The first holds because T_* initially contains the unique method for decomposing τ_{s} that must appear in all decompositions. The second holds because T only has the two distinguished start and stop tasks appearing in all action sequences. The third holds because O's initial elements are also in O^{s} implying that any π consistent with O^{s} is also consistent with O. Finally, the last holds because the causal constraint $\{\langle\rho@\infty\rangle\}$ must match the causal support for the end of any solution sequence. $\square$

6.2 Common Steps

In order to reduce effort when proving the recursive invariants for both UCPARSE and UCREDUCE, we point out that the algorithms share several contiguous sequences of steps, prove properties on these components, and combine these properties to prove the invariants.

6.2.1 Termination

To show how the two soundness invariants combine to prove that UCPARSE and UCREDUCE return action sequences that solve the FHTN planning problem, consider the termination step in figure 6.1. This step is equivalent to the termination steps found in each planner. While the equivalence for UCPARSE stems from the fact that all tasks in T are always primitive, the equivalence for UCREDUCE stems from *agenda* never having selectable open conditions when testing termination.

1. **Termination:** When T has no abstract tasks and *agenda* has no selectable open conditions: fail if $D_* \neq \emptyset$, otherwise succeed and return $\texttt{sol}(\langle T, B, O, L\rangle)$.

Figure 6.1: Common step used to return a solution

Although the termination step does not directly test for an empty *agenda*, it must be empty whenever calling $\texttt{sol}()$. By definition, an open condition $\langle Q@n\rangle$ is selectable if τ_n does not appear in D_*. Since D_* must be empty prior to the call, so must *agenda*.

Lemma 4 *The call* $\texttt{sol}(\langle T, B, O, L\rangle)$ *returns a sequence* $\{a_i\}_{i=0}^m$ *solving problem* $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho\rangle$ *whenever* $\langle T, B, O, L\rangle$ *is primitive, and the partial plan is both goal and task oriented with empty* D_* *and methods instantiated from* $\mathcal{M}$.

Proof: By definition, $\texttt{sol}(\langle T, B, O, L\rangle)$ nondeterministically orders the tasks and grounds variables to return an action sequence $\{a_i\}_{i=0}^m$ such that $(n{:}\texttt{do[}a_{\pi(n)}\texttt{]}) \in T\backslash\theta$, where π and θ respectively define the ordering and grounding substitution. The only constraints on π and θ are that $\pi(n) < \pi(m)$ whenever $(\tau_n \prec \tau_m) \in O$ and $B\backslash\theta$ holds.

Now since the partial plan is goal oriented with an empty agenda, we have

$$\begin{aligned}
\textsc{dnf}(assured_n, \mathcal{I}) &\subseteq \textsc{dnf}(required_n, \mathcal{I});\\
\textsc{dnf}\left(\textstyle\bigwedge_{m\xrightarrow{\ell}n\in L} \ell, \mathcal{I}\right) &\subseteq \textsc{dnf}(required_n, \mathcal{I});\\
\{\{\ell \mid m\xrightarrow{\ell}n \in L\}\} &\subseteq \textsc{dnf}(required_n, \mathcal{I});\ \text{and finally}\\
\{\ell \mid m\xrightarrow{\ell}n \in L\} &\in \textsc{dnf}(required_n, \mathcal{I}).
\end{aligned}$$

Given that D_* is empty we substitute for $required_n$ and $preserved_n$ to get

$$\{\ell \mid m\xrightarrow{\ell}n \in L\} \in \textsc{dnf}\left(pre(a_{\pi(n)}) \wedge \left(\bigwedge_{n\xrightarrow{\ell}m\in L} \Sigma_\ell^{a_{\pi(n)}}\right) \wedge \left(\bigwedge_{m\xrightarrow{\ell}o\in L} \Pi_\ell^{a_{\pi(n)}}\right), \mathcal{I}\right),$$

where the ordering constraints in O imply $(m \prec n \prec o)$. This equation and `sol`()'s getting the action sequence by ordering tasks in T implies that the returned action sequence is goal directed toward ρ with

$$L' = \{\pi(m) \xrightarrow{\ell} \pi(n) \mid m \xrightarrow{\ell} n \in L\}.$$

In order to prove task directedness, notice that the partial plan's task orientation implies that for any grounding substitution θ

$$\tau_{\mathtt{s}} \overset{*}{\Rightarrow} \langle \emptyset : T^\theta, B^\theta, O^\theta, L^\theta : \emptyset \rangle,$$

where the 5 conditions hold. Given that $T \backslash \theta = T^\theta$ and T only has primitive tasks, we see that $\langle \emptyset : T^\theta, B^\theta, O^\theta, L^\theta : \emptyset \rangle \in \mathcal{L}(\mathcal{M})$ for any θ satisfying B. Since `sol`$(\langle T, B, O, L \rangle)$ returns an action sequence that performs $\langle T, B, O, L \rangle$ with a grounded variables, this action sequence also performs a member of $\mathcal{L}(\mathcal{M})$.

Finally, the justification property falls directly from the fifth task orientation condition combined with D_* being empty. □

This next lemma shows that maintaining the third invariant guarantees that our algorithms can continue making progress toward a chosen solution until `sol`() is guaranteed to be able to return the solution sequence. This guarantee is needed to prove completeness.

Lemma 5 *If the partial plan is consistent with a solution sequence and its decomposition of τ_s, then either the task network has an abstract task, the agenda has a selectable open condition, or* `sol`() *can return the solution sequence.*

Proof: Given an execution of the termination step, suppose that all tasks are primitive, *agenda* lacks selectable flaws, and the termination step cannot return the solution sequence; either one of three cases holds: (1) the step fails due to D_* containing elements or either (2) some action missing from T or (3) some inconsistency between O and B and π and θ^{s} implies that `sol` cannot produce the solution sequence. While the third case directly contradicts the consistency between the partial plan's tasks and $\{a_i\}_{i=0}^{k}$, we show that the first two cases contradict the consistency property's matching between causal constraints and causal supports.

From our algorithms' definitions, D_* can only contain unjustified primitive tasks and causal constraints. Given UCPARSE's definition and UCREDUCE's definition combined with the sound nonprimitive task requirement, we know that D_* can only contain unjustified primitive tasks when performing the termination step. Given this result, the first two cases respectively apply when a task is unjustified or missing.

Let a denote the latest solution-sequence action either lacking a matching task in T or having τ_n also present in D_*. Since there are no selectable open conditions, a must either denote or succeed the latest action with a task having an open condition. If some later task had a non-selectable open condition, then it would be in D_* and a would denote its action.

Since a solution-sequence action must have a justifying link, some causal support for a later action either lacks a matching causal constraint or that constraint has an open condition. If the constraint is missing, then the task or link motivating the constraint is missing implying an open condition for a later task. Therefore having no selectable open conditions implies the contradiction that a is not the latest solution-sequence action either lacking a matching task in T or having τ_n also appearing in D_*. □

6.2.2 Resolving Non-Literal Open Conditions

In figure 6.2 we show the exact step used by both planners to breakup a non-literal open condition into its components. This step computes the non-literal's disjunctive normal form and then nondeterministically chooses a disjunct. Since this disjunct can have both variable constraints and literal conditions, the step adds the literals to *agenda* and the constraints to B.

While this step does affect sets in a partial plan, the following two lemmas prove

1. **Non-literal:** If Q_k is not a literal, nondeterministically choose a conjunct Γ from the set $\text{DNF}(Q_k, \mathcal{I})$, remove $\langle Q_k@n_k\rangle$ and add $\langle \ell@n_k\rangle$ to *agenda* for each literal $\ell \in \Gamma$, and add Γ's constraints to B– failing on inconsistent B;

Figure 6.2: Common step for resolving non-literal open conditions

that breaking up a non-literal open condition violates neither task nor goal orientation.

Lemma 6 *A partial plan's goal orientation is preserved when resolving non-literal open conditions.*

Proof: Given the non-literal open condition $\langle Q_k@n_k\rangle$, both algorithms take the results of calling $\text{DNF}(Q_k, \mathcal{I})$ and alter the *agenda* and task network the same way. In this case only $assured_{n_k}$ changes to reflect changes to goals involving τ_{n_k}. By defining $assured'_{n_k}$ to be the result of removing $\langle Q_k@n_k\rangle$ from *agenda* and choosing any Γ from $\text{DNF}(Q_k, \mathcal{I})$, we get:

$$\begin{aligned}
\text{DNF}(assured_{n_k}, \mathcal{I}) &= \text{DNF}(Q_k \wedge assured'_{n_k}, \mathcal{I}) \\
&= \{\phi_1 \cup \phi_2 \mid \phi_1 \in \text{DNF}(Q_k, \mathcal{I}) \text{ and } \phi_2 \in \text{DNF}(assured'_{n_k}, \mathcal{I})\} \\
&\supseteq \{\Gamma \cup \phi_2 \mid \phi_2 \in \text{DNF}(assured'_{n_k}, \mathcal{I})\} \\
&\supseteq \text{DNF}((\textstyle\bigwedge_{Q \in \Gamma} Q) \wedge assured'_{n_k}, \mathcal{I}) \\
&\supseteq \text{DNF}(assured''_{n_k}, \mathcal{I}),
\end{aligned}$$

where $assured''_{n_k}$ results from removing $\langle Q_k@n_k\rangle$, adding $\langle Q@n_k\rangle$ to *agenda* for each literal $Q \in \Gamma$, and adding Γ's constraints to B_{n_k}. Given that $assured''_{n_k}$ denotes the conditions assured to be true for τ_{n_k} after resolving the non-literal open condition, goal orientation is maintained because

$$\text{DNF}(assured''_{n_k}, \mathcal{I}) \subseteq \text{DNF}(assured_{n_k}, \mathcal{I}) \subseteq \text{DNF}(required_{n_k}, \mathcal{I}).\Box$$

Lemma 7 *The partial plan's task orientation is preserved when resolving non-literal open conditions.*

Proof: By definition, task orientation depends on a property holding for every grounding substitution θ that satisfies B. Since resolving non-literal open conditions only changes *agenda* and adds constraints to B, the only effect on task orientation involves a reduction of the substitutions satisfying B. Therefore a partial plan's task orientation is preserved. □

Unlike the previous 2 lemmas this next lemma is concerned with completeness. The objective here is to maintain consistency with a predetermined solution while resolving the nonliteral open condition. One important property that this lemma takes advantage of involves the soundness of ADL actions that model abstract tasks. All of the preconditions of such an action are guaranteed to be required of a subtask of the action. This implies that any nonliteral open condition for an abstract task must be a conjunct, and that this step does not have a nondeterministic choice.

Lemma 8 *If the partial plan is consistent with a solution sequence and its decomposition of τ_s, then there exists a nondeterministic choice such that resolving a nonliteral open condition maintains the consistency.*

Proof: Given that resolving a nonliteral open condition $\langle Q_k @ n_k \rangle$ only affects a partial plan's binding constraints and *agenda*, we need only consider preserving the grounding substitution $\theta^{\mathbf{s}}$'s validity and the match between causal constraints and the solution's causal supports.

Consider those steps in UCPARSE and UCREDUCE that add nonliteral open conditions for primitive actions; in all cases these conditions assure that an action can either execute, provide a desired effect, or avoid producing an undesired effect. Since the solution sequence's definition requires a causal support with a set of links and variable constraints assuring an element of $\textsc{dnf}(Q_k)$ at τ_{n_k} there exists a nondeterministic choice that preserves both θ^{s} and the match between the partial plan's causal constraints and the solution's causal supports.

Now suppose that τ_{n_k} denotes an abstract task. Neither Q_k nor its DNF breakdown participates as a causal constraint. If Q_k has a variable constraint, the soundness requirement for ADL actions modeling abstract tasks implies that the same variable constraint must be added by some primitive component task to assure that it can execute to provide some effect. For this reason θ^{s} is not invalidated. $\square$

6.2.3 Tracing Derivations

In figure 6.3 we show the exact steps used by both algorithms to trace a derivation in order to determine how to extend the task network to support an open condition. The main source of complexity here derives from a least commitment approach to deciding how to ground variables and which tasks to include in a reduction; this approach makes these steps reason about elements drawn from the set $\mathcal{M}$ of variablized methods instead of the larger grounded set $G(\mathcal{M})$.

1. **Derivation traversal:** While ascending and descending, let $e \overset{b}{\in} S$ denote that there exists $e' \in S$ and $e \overset{B}{=} e' \backslash b$, where b denotes a substitution for universal variables.
   - (a) **Ascend:** Starting with $n = n_k$, repetitively let $n = a(n)$ while $(\overset{Q_k}{\rightarrow} n) \overset{b}{\in} P_{a(n)}$ in T_*. Ultimately let $n = m$, where $(m \overset{Q_k}{\rightarrow} n) \overset{b}{\in} L_{a(n)}$ in T_*.
   - (b) **Descend:** Repetitively let $n = m$ while $(m \overset{Q_k}{\rightarrow}) \overset{b}{\in} E_n$ in T_*.
2. **Universal variable resolution:** If n contains variables and $(T \cup S_*)$ lacks τ_n and (n', n), take all $[\tau_m \Rightarrow ctn_m]$, τ_m, or (m', m) from $(T_* \cup T \cup S_*)$, where $m \overset{b}{\cong} n$, and nondeterministically choose one or none. For those cases where one was chosen, add b to B and resume descend; Otherwise, add $O^n_{a(n)}$ to O and for each element add $m \ncong n$ to B.

Figure 6.3: Common steps used to trace a precondition through a derivation

As shown in the figure, tracing a condition through a derivation adds ordering constraints of the form $O^n_{a(n)}$ during universal variable resolution. These denote elements of the universally quantified constraints that just begin to apply to the task network due to discovering the need for a new task τ_n, from a universally quantified set, such that either $(n \overset{Q_k}{\rightarrow} o) \overset{b}{\in} L_{a(n)}$ or $(n \overset{Q_k}{\rightarrow}) \overset{b}{\in} E_{a(n)}$. These added constraints explicitly restrict the newly discovered task as required by the decomposition method.

Given that these steps only add ordering and binding constraints. Proving that both goal and task orientation are maintained is straight forward.

Lemma 9 *Both goal and task orientation are preserved when tracing derivations.*

Proof: As figure 6.3 shows, the B and O are the only sets to change while tracing a derivation, and all changes only add constraints. For this reason, each equation $\textsc{dnf}(assured_n, \mathcal{I}) \subseteq \textsc{dnf}(required_n, \mathcal{I})$ remains unaffected while tracing a derivation, and goal orientation is preserved.

Since adding to B only reduces the number of acceptable grounding substitutions, and adding to O has no impact on the existence of $d_i^\theta \in \mathcal{G}^*(\{d_i\})$ such that

$$\tau_{\mathbf{s}} \overset{d_1^\theta}{\Rightarrow} ctn_1 \overset{d_2^\theta}{\Rightarrow} \ldots \overset{d_k^\theta}{\Rightarrow} \langle \emptyset : T^\theta, B^\theta, O^\theta, L^\theta : \emptyset \rangle,$$

the $d^\theta_{\{1\ldots k\}}$ methods used prior to tracing a derivation still apply for a grounding substitution θ that still satisfies B after tracing; thus task orientation is preserved. $\Box$

While the previous lemma is used to prove soundness, our next lemma facilitates the completeness proofs.

Lemma 10 *If a partial plan is consistent with a solution sequence's decomposition, then there exists a set of nondeterministic choices that maintains consistency while tracing a derivation.*

Proof: Since the derivation traversal step adds no constraints, it cannot invalidate consistency. Unlike derivation traversal, the universal variable resolution step can add ordering and variable constraints. These constraints make n denote a particular element of a universally quantified set of tasks (or point protections). In adding variable constraints, there is a nondeterministic choice to unify a variablized label n with one k labels, or invalidating the possibility of unifying with any of them. Since this choice covers all possible cases, there is guaranteed to be a nondeterministic choice that does not invalidate θ^{s}.

With regards to adding the ordering constraints when differentiating n from the k other labels, these ordering constraints are universally applied to all tasks with labels like $n\backslash\theta^{\mathsf{s}}$. As such, these ordering constraints must already be in O^{s}, and cannot affect the consistency between O and π when choosing to make n unique. $\square$

In addition to maintaining goal and task orientation and possibly consistency with some solution, these steps compute another property required to assure that steps subsequently used to resolve the literal open condition maintain task orientation. These steps add variable binding constraints such that the derivation prescribes the same repair regardless of how the variables are grounded.

Definition 15 *A derivation $\langle\{d_1 \ldots d_k\}, S_*, D_*\rangle$* PRESCRIBES *open condition $\langle Q@n\rangle$'s support if there exists a label m such that for all θ satisfying B a set of decompositions $d^{\theta}_{\{1..k\}}$ can only produce support for Q at τ_n in the form of the auxiliary constraint $(m \xrightarrow{Q} n)\backslash\theta$, where m either refers to no element or the same element of $T \cup S_*$ for each θ.*

The main point to emphasize here is that the label m must be the only prescribed label for any θ satisfying B. This implies a constraint that there can never be two elements in either $(L_n \cup P_n)$ denoting how to support a point protection $\langle Q@m\rangle$. Also no E_n can assign more than one subtask of τ_n to provide a condition Q. If such a case did occur, determining how to support a condition becomes ambiguous.

Lemma 11 *If no set S from $(L_n \cup P_n)$ or E_n in any derivation method has more than one element $e \overset{b}{\in} S$ supporting a given open condition, the derivation prescribes $\langle Q_k@n_k\rangle$'s support with label n after tracing.*

Proof: By the "$\Rightarrow$" relation's definition, support for $\langle Q_k@n_k\rangle\backslash\theta$ appears in L^θ when three conditions hold: (1) there exists a d^θ_{top} with $(m^\theta_{top})\overset{Q_k\backslash\theta}{\rightarrow}(n^\theta_{top})$ in L^θ_{top}; (2) other $d^\theta_{\{1\ldots k\}}$ methods replace m^θ_{top} with n^θ and n^θ_{top} with $n_k\backslash\theta$; and (3) no methods decompose tasks labeled by n^θ or $n_k\backslash\theta$.

Given the ancestor function[1], the ascend and descend steps determine how to alter the $d^\theta_{\{1\ldots k\}}$ methods by finding the unique elements:

1. $\overset{Q_k}{\rightarrow}a^i(n_k) \overset{b_i}{\in} P_{a^{i+1}(n_k)}$ for i ranging from zero to $x-1$;

2. $a^y(n)\overset{Q_k}{\rightarrow}a^x(n_k) \overset{b_x}{\in} L_{a^{x+1}(n_k)}$; and

3. $a^j(n)\overset{Q_k}{\rightarrow} \overset{b_{j+x+1}}{\in} P_{a^{j+1}(n)}$ for j ranging from zero to $y-1$.

As the found elements show, the most common relationship used while tracing a derivation is $e \overset{b}{\in} S$ for some component S of a contextualized task network. By definition, the universal variable expansion $\mathcal{U}(\{[\tau{\Rightarrow}ctn]\})$ computes a singleton set with one method, and each set S in ctn is replaced with $\{e \mid e \overset{b}{\in} S\}$, where b is a grounding substitution for universal variables. Since $\mathcal{G}$ grounds the existential variables, each grounding substitution θ denotes a unique method in $\mathcal{G}(\mathcal{U}(\{[\tau{\Rightarrow}ctn]\}))$, and whenever $e \overset{b}{\in} S$ for a set in ctn, $e\backslash\theta$ is in the corresponding set in $\mathcal{G}(\mathcal{U}(\{[\tau{\Rightarrow}ctn]\}))$'s method for θ. Finally, $\mathcal{G}^*(\{[\tau{\Rightarrow}ctn]\})$ uses $\mathcal{G}$ and $\mathcal{U}$ to compute a method for each θ and then returns a set of methods for each θ by taking subsets of the computed method's components P, T, L, and E. Thus finding $e \overset{b}{\in} S$ for some set in $[\tau{\Rightarrow}ctn]$ implies that $e\backslash\theta$ can be added to the corresponding set in any $d^\theta_i \in \mathcal{G}^*(\{[\tau{\Rightarrow}ctn]\})$. Thus the ascend and descend steps find n such that for all θ satisfying B the $d^\theta_{\{1\ldots k\}}$ methods

[1] Function $a^i(n)$ denotes applying the ancestor function i times, and $a^0(n)$ is an identity.

can be modified to add $n \xrightarrow{Q_k} n_k \backslash \theta$ to L^θ if for each θ there is no decomposition for $\tau_n \backslash \theta$ in $d^\theta_{\{1 \ldots k\}}$.

If n contains variables and $(T \cup S_*)$ lacks both τ_n and (n', n), n may or may not denote any of a possibly empty set of previously defined tasks or point protections. From figure 6.3 we see that the universal variable resolution restricts the substitutions satisfying B to those that make n refer to either none or a particular previously defined task or point protection. If none was chosen the lemma holds because n cannot refer to a task in T. If one was chosen the descend step is resumed to assure its termination property. Thus the descend step's loop invariant is maintained and universal variable resolution does not pass control on until either none was chosen, n contains no variables, or $(T \cup S_*)$ contains τ_n or (n', n).

Thus three cases hold after universal variable resolution. In one case, none was chosen making n unable to denote a point protection or task in $(T \cup S_*)$. In the second and third cases n always denotes the same task or point protection because either (1) τ_n or (n', n) appear in $(T \cup S_*)$ or (2) n is unaffected by any substitution by virtue of lacking variables. In either case the descend step's termination implies that $\tau_n \backslash \theta$ is not decomposed for any θ, and the lemma holds. $\square$

6.2.4 Resolving a Literal Open Condition

In figure 6.4 we show the exact steps used to support a literal open condition with a causal link given a prescribed auxiliary constraint. While the last three steps are similar to those that would appear in any partial order planner, the first step is devoted to maintaining completeness in the face of point protections – a feature commonly used in HTN planning. In this step, newly found point protections are resolved by either linking to existing or newly added primitive tasks. The set D_* contains those tasks that only support point protections; it can acquire new elements when supporting with new point protections and loose elements as the algorithms support with new primitive tasks.

1. **Point protection:** Handle point protections by:
   (a) if $(m, n) \in S_*$, then let $n = m$;
   (b) if n is a new point protection, nondeterministically choose $\tau_m = (m{:}\mathtt{do[}a\mathtt{]})$ from a set of existing (from T) or new (instantiated from $\mathcal{O}$) tasks that can assert Q_k, add (m, n) to S_*, add τ_m to D_* if τ_m is new, and finally let $n = m$;
   (c) if n is a task label denoting a new task $(n{:}\mathtt{do[}a\mathtt{]})$, nondeterministically choose $(n'{:}\mathtt{do[}a'\mathtt{]}) \in D_* \cup \{\tau_n\}$ such that $a \overset{b}{\cong} a'$. Let $B = B \cup b$, remove $(n'{:}\mathtt{do[}a'\mathtt{]})$ from D_*, and substitute n for n' in T, O, L, S_*, D_*, and *agenda*.
2. **Satisfaction:** Add $(n \xrightarrow{Q_k} n_k)$ to L, add τ_n to T if new, and add $(n \prec n_k)$ to O when both τ_n and τ_{n_k} are primitive (fail if O becomes inconsistent).
3. **Subgoal generation:** Where τ_n executes a, alter *agenda* to remove $\langle Q_k @ n_k \rangle$, add $\langle \mathit{pre}(a) @ n \rangle$ when τ_n new, and add $\langle \Sigma^a_{Q_k} @ n \rangle$ when $\langle n | Q_k \rangle \notin D_*$ – else remove $\langle n | Q_k \rangle$.
4. **Causal link protection:** Primitive task τ_l *threatens* a causal link $n \xrightarrow{\ell} m$ when it asserts $\neg\ell$ between primitives τ_n and τ_m. For each link $n \xrightarrow{\ell} m \in L'$ and each $(l{:}\mathtt{do[}a\mathtt{]})$ that might threaten it, nondeterministically choose to:
   (a) **Promote:** Add $(m \prec l)$ to O, and fail if inconsistent.
   (b) **Demote:** Add $(l \prec n)$ to O, and fail if inconsistent.
   (c) **Confront:** Add $(n \prec l)$ and $(l \prec m)$ to O, and $\langle \neg \Sigma^a_{\neg\ell} @ n \rangle$ to *agenda*.

Figure 6.4: Common steps used to resolve a literal open condition

The main difference between the last three steps and those in a partial order planner reflects the derivation's prescribing how to support the open condition; partial order planners typically use nondeterministic choice to determine which task supports a given open condition.

The subgoal generation step also differs slightly due to the use of elements like

$\langle n|Q_k\rangle$ in D_*; this difference is also apparent when comparing this step with the subgoal generation step in UCPARSE. Each element $\langle n|Q_k\rangle$ records that task τ_n had a subgoal generated for supporting a condition Q_k but currently lacks the causal link denoting how the subgoal is used. While these elements never appear for UCPARSE, they are created during reduction in UCREDUCE. When we drop the references to these elements, the subgoal generation step turns into the one appearing in UCPARSE's definition.

Lemma 12 *If a partial plan is goal oriented prior to the point protection step, it is goal oriented after the causal link protection step.*

Proof: The point protection steps consists of three cases to respectively reuse old point protections, create new point protections, and reuse newly created primitive tasks to satisfy old point protections. While the first two cases have no effect on goal orientation, the third globally substitutes a label n for n' within the partial plan. Since $(n{:}\texttt{do[}a\texttt{]})$ is a newly created primitive with the same action as $(n'{:}\texttt{do[}a\texttt{]})$ the fact that a particular goal orientation equation holds is invariant because $\textsc{dnf}(assured_{n'}) \subseteq \textsc{dnf}(required_{n'})$ implies that $\textsc{dnf}(assured_n) \subseteq \textsc{dnf}(required_n)$ after the global substitution. Thus the point protection step preserves a partial plan's goal orientation.

To prove that goal orientation holds after performing the point protection, satisfaction, subgoal generation, and causal link protection steps, note that causal link protection makes $\langle T, B, O, L\rangle$ threatless and consider each goal orientation equation $\textsc{dnf}(assured_m, \mathcal{I}) \subseteq \textsc{dnf}(required_m, \mathcal{I})$; we need to look at four cases depending on whether m denotes τ_n, τ_{n_k}, or neither.

1. If m denotes τ_n, two subcases apply depending on whether or not τ_n was just inserted into T. For old tasks, the equation still holds because both $required_m$ and $assured_m$ get a new conjunct $\Sigma^a_{Q_k}$, where a is τ_n's action. For new tasks, the equation holds because $required_m$ equals $assured_m$ due to explicitly adding all of $required_m$'s conjuncts to the *agenda*.

2. If m denotes τ_{n_k}, the goal orientation constraint is preserved because neither $required_m$ or $assured_m$ changes. The only change involves moving Q_k from one place to another in the $assured_m$ conjunct to reflect removing $\langle Q_n@n_k\rangle$ from $agenda \cup \{\langle Q_n@n_k\rangle\}$ and adding $n\xrightarrow{Q_k}n_k$ to L.

3. If m denotes neither τ_n or τ_{n_k}, $required_m$ might acquire extra conjuncts due to adding to $preserved_m$ while confronting threats. The goal orientation constraint is preserved because each new acquisition is also reflected in $assure_m$ by adding the same goal to $agenda$.

Finally, since each goal orientation constraint holds and the task network is threatless, goal orientation holds for the partial plan after the causal link protection step. □

Now that goal orientation holds, we take advantage of the prescription property to prove that resolving a literal open condition maintains task orientation.

Lemma 13 *If a partial plan is task oriented prior to the point protection step and the derivation prescribes $\langle Q_k@n_k\rangle$'s support with label n, then task orientation holds after the causal link protection step.*

Proof: Due to the prescription property, we know that for all θ the derived task network can be altered to contain $(n\xrightarrow{Q_k}n_k)\backslash\theta$ in L^θ; we also know that regardless of θ, $n\backslash\theta$ either never denotes a task in $T\backslash\theta$ or always denotes $\tau\backslash\theta$ for the same $\tau \in T$.

Now consider figure 6.4 and the prescribed support n denoting either a new or previously-seen task or point protection, we need to consider four cases depending on n. While all cases can add ordering constraints, these additions preserve task orientation because $O^\theta \subseteq O\backslash\theta$ is unaffected. Similarly, add binding constraints to B does not affect task orientation.

This first and second cases respectively involve existing tasks and point protections. When τ_n already appears in T; here the only changes to the task network or derivation involve adding $n \xrightarrow{Q_k} n_k$ to L and possibly some ordering constraints to O. Adding the link does not affect task orientation due to the prescription property. When n is a previously seen point protection signified by $(m, n) \in S_*$, the only changes to the task network or derivation involve adding $m \xrightarrow{Q_k} n_k$ to L and possibly some ordering constraints to O. Once again, adding the link does not affect task orientation due to the prescription property and the way that point protections are handled using $S_* \backslash \theta$ in the task orientation definition.

When n denotes a new task, there are two subcases depending on whether or not the point protection step recycles a task from D_* to serve as the new task τ_n. Both subcases require adding $n \xrightarrow{Q_k} n_k$ to L, τ_n to T, and possibly some ordering constraints to O. None of these additions affect task orientation due to the prescription property, and task orientation is preserved when adding new tasks. In the case where a task is recycled, task orientation is still preserved because n's being unequal to any label currently in T or S_* implies that globally substituting n for n' has no effect on task orientation; neither does the added binding constraints. Finally the task τ_n removed from $D_* \backslash \{(n', n)\}$ can now appear in T^θ due to the possibility of $(n \xrightarrow{Q_k} n_k) \backslash \theta \in L^\theta$.

In the fourth, and last, case n denotes a new point protection. Once again there are two subcases depending on whether or not a task is specifically added to support the point protection. Both cases add (m, n) to S_*, $m \xrightarrow{Q_k} n_k$ to L, and possibly some ordering constraints to O. Just like the case involving existing point protections, none of these additions affect task orientation. When the two subcases differ, task orientation is preserved when adding a new task to T because it is also added to D_*. $\square$

Just like the previous lemma, this next proof takes advantage of the prescription property to prove that making the appropriate nondeterministic choices results in adding constraints that do not invalidate the consistency of a partial plan with a designated solution.

Lemma 14 *There exists a set of nondeterministic choices that maintains consistency with a solution sequence and its decomposition while resolving a literal open condition whenever the derivation unambiguously prescribes a task or point protection with label* $n\backslash\theta$ *for all* θ *and all ADL actions modeling abstract tasks are sound.*

Proof: Given the prescription property, we know that a particular label $n\backslash\theta^{\mathsf{s}}$ denotes the task supporting $Q_k\backslash\theta^{\mathsf{s}}$ for task $n_k\backslash\theta^{\mathsf{s}}$. With regards to our consistent solution sequence and its decomposition of τ_{s}, this and the soundness of ADL actions modeling abstract tasks implies that L^{s} has some link $n' \xrightarrow{Q_k} n'_k$, where n' and n'_k are either $n\backslash\theta^{\mathsf{s}}$ and $n_k\backslash\theta^{\mathsf{s}}$ or labels appearing in the full decompositions of $\tau_n\backslash\theta^{\mathsf{s}}$ and $\tau_{n_k}\backslash\theta^{\mathsf{s}}$ respectively.

With this in mind, consider the first step in figure 6.4. Proving that the point protection step can always maintain consistency involves taking π with the solution's link $n' \xrightarrow{Q_k} n'_k$ and showing that the step can always replace the prescribed label n with a label consistent with n' and π. This involves addressing three cases depending on the prescribed label's denoting an abstract task, primitive task, or point protection. For abstract tasks, the first step is guaranteed to maintain consistency because it always leaves n unchanged. For point protections, $\pi(n')$ denotes an action whose primitive task may or may not already exist in T. The first step can maintain consistency in either case because it nondeterministically uses an existing abstract task or newly created task. On the other hand, when n denotes a new primitive task $\pi(n')$ denotes a primitive task that might have been previously added to satisfy a point protection. Once again consistency is assured by nondeterministically choosing to add or use an existing unjustified task.

Once the point protection step updates n to be consistent with π and the solution's link $n' \xrightarrow{Q_k} n'_k$, the satisfaction and subgoal generation steps are guaranteed to add links, tasks, ordering constraints and subgoals that are consistent with the solution sequence. This guarantee is due to the solution's requiring these elements for soundness.

Finally, the last step of figure 6.4 adds ordering constraints and subgoals to protect causal links between primitive tasks from other primitive tasks. Once again consistency can be maintained because each protection involves nondeterministically choosing to add ordering constraints that put the threatening task either before the link, during the link, or after the link. Since the covers all times, there exists a choice that maintains consistency. If the threatening step is ordered during the link, the added open condition cannot invalidate consistency due to the requirement for soundness. □

6.3 Algorithms

Using the previously described sequences of steps, we now define our FHTN planning algorithms and prove their soundness and completeness.

6.3.1 UCPARSE

As figure 6.5 illustrates, UCPARSE selects and repairs one open condition per recursive call. The termination step detects cases where there are no selectable conditions and either succeeds of fails. If a condition is selected, it is resolved either as a literal or a nonliteral.

The point where this description differs from the earlier one revolves around embedding the point protection step in the resolve open condition sequence. In the earlier characterization the point protection step was executed regardless of the condition type. Since resolving a non-linear task adds neither tasks not causal links, it does not cause any threats to resolve, and the protection step does nothing. This implies that we can move the step into the resolve open condition sequence without changing the algorithm.

Using the previous results we now prove UCPARSE's soundness by first proving

Algorithm UCPARSE($\langle T, B, O, L\rangle, \langle T_*, S_*, D_*\rangle, agenda$)

1. **Termination:** see figure 6.1
2. **Open condition resolution:** Select $\langle Q_k@n_k\rangle$ (not referring to a member of D_*) from *agenda*, and resolve it with one of the two cases:
   (a) **Non-literal:** See figure 6.2
   (b) **Literal:** If Q_k is a literal, either fail if $m \xrightarrow{\neg Q_k} n_k \in L$ (an impossible plan) or:
      i. **Trace derivation:** see figure 6.3
      ii. **Reduction:** While n denotes abstract task τ_n: nondeterministically choose/add a new (instantiated from $\mathcal{M}$) method $[\tau_n \Rightarrow ctn_n]$ to T_*; find $(m \xrightarrow{Q_k}) \overset{b}{\in} E_n$; add B_n to B; add $O_n^{\emptyset} \cup O_n^m$ to O; and finally let $n = m$.
      iii. **Resolve open condition:** see figure 6.4
3. **Recursive call:** Call UCPARSE($\langle T, B, O, L\rangle, \langle T_*, S_*, D_*\rangle, agenda$)

Figure 6.5: Structuring common steps into UCPARSE

that goal and task orientation are recursive invariants. With these results we prove soundness with an induction argument over UCPARSE's recursive calls to show that goal and task orientation always holds. Given this result lemma 4 implies that the termination step will only succeed and return an action sequence if the sequence forms a solution to the problem.

Lemma 15 *A partial plan's goal orientation is a recursive invariant for* UCPARSE.

Proof: Given that the termination step does not affect any of these items, this proof has two cases depending on whether or not the open condition resolved is a non-literal. If it is a non-literal, lemma 6 implies that the goal orientation is true at the recursive call if it was true upon entering UCPARSE.

In the case where the open condition resolved is a literal, lemmas 9 and 12 respectively imply that tracing the derivation and resolving the open condition preserve goal orientation. While the reduction adds to T_*, B, and O, none of these additions affect goal orientation. Since nothing from T_* and only subsets of B participate in the goal orientation equations, adding to T_* and B never affects a primitive task's goal orientation equation. Since all ordering constraints added to O involve labels of subtasks of τ_n, none of them affect tasks outside of τ_n. Since all goal orientation equations involve existing primitive tasks outside of τ_n, no equation is affected. □

Lemma 16 *A partial plan's task orientation is a recursive invariant for* UCPARSE.

Proof: Given that the termination step does not affect any of these items, this proof has two cases depending on whether or not the open condition resolved is a non-literal. If it is a non-literal, lemma 7 implies that the partial plan is task oriented at the recursive call if it was upon entering UCPARSE.

From lemmas 9 and 11 we know that task orientation is preserved while tracing a derivation, and afterwards the derivation prescribes τ_n to support $\langle Q_k @ n_k \rangle$. In the reduction step, task orientation is preserved due to the way elements are added to B

and O. Prescription is also maintained due to the way E_n is used to alter n. Since both of these properties hold upon starting to resolve the open condition, lemma 13 implies a task oriented partial plan prior to the recursive call. Thus task orientation is a recursive invariant. $\Box$

Theorem 1 (Soundness) *Given $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$ any action sequence returned by the initial call to* UCPARSE *solves $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$.*

Proof: Given that lemmas 1 and 2 imply that goal and task orientation hold for the initial call, an induction argument using the recursive invariants imply that these properties hold both for every call and whenever the termination step executes; now consider when this step calls `sol`().

Since T has no abstract tasks, the task network is primitive. Since D_* is empty, and *agenda* has no selectable tasks, *agenda* must also be empty. At this point, these properties combine with lemma 4 to imply that `sol`() returns a solution to the problem whenever UCPARSE succeeds. $\Box$

We now turn our attention to UCPARSE's completeness. To prove this property we hypothesize some arbitrary solution sequence $\{a_i\}_{i=0}^m$ and use the previous consistency lemmas to prove that UCPARSE can find it.

Theorem 2 (Completeness) *Let $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$ be a planning problem with a solution sequence $\{a_i\}_{i=0}^m$. The initial call to* UCPARSE *will return the sequence given an appropriate set of nondeterministic choices.*

Proof: Consider a solution sequence $\{a_i\}_{i=0}^m$, from lemma 3 we know that the sequence is consistent with the initial call's partial plan. Given that the sequence is consistent with a recursive call, lemma 5 implies that either planning can continue to the next recursive call, or that SOL can take the partial plan and return the solution sequence.

Now suppose that planning can continue to the next recursive call. Lemma 8 combined with the fact that the termination step adds no constraints implies that there exists a nondeterministic choice that maintains consistency to the next recursive call when resolving a nonliteral open condition.

If the selected open condition is literal, lemma 10 implies that consistency is maintained through to the reduction step. Since the reduction step only alters n when it denotes an abstract task, and only through using a nondeterministically chosen decomposition method, the reduction step can maintain consistency while assuring that n still denotes the prescribed (see lemma 11) way to resolve the open condition. This prescription combined with lemma 14 implies that UCPARSE can maintain consistency while resolving a literal open condition.

Finally, each call to UCPARSE either resolves a nonliteral open condition or adds a causal link. Since each nonliteral added to the plan is a result of adding a task or causal link, the number of nonliteral and literal open conditions resolved while producing the solution must be finite. Thus lemma 5 implies that UCPARSE must eventually terminate by calling SOL to return the final solution. □

6.3.2 UCREDUCE

As figure 6.6 illustrates, UCREDUCE selects and repairs one open condition per iteration of its critic loop. The termination step detects cases where there are no abstract tasks and either succeeds of fails. While abstract tasks exist, each recursive call selects and reduces one.

Unlike UCPARSE, UCREDUCE can break down and resolve many open conditions per recursive call. This difference drives changes in our completeness proofs; we first prove that the two properties are loop invariants for the critic loop and then prove their recursive invariance. Finally their recursive invariance leads to the soundness proof.

Algorithm UCREDUCE$(\langle T, B, O, L\rangle, \langle T_*, S_*, D_*\rangle, agenda)$

1. **Criticism:** While *agenda* has selectable open conditions, select $\langle Q_k@n_k\rangle$ (not referring to a member of D_*) from *agenda*, and resolve it with one of the two cases:
   (a) **Non-literal:** See figure 6.2
   (b) **Literal:** If Q_k is a literal, either fail if both $m \xrightarrow{\neg Q_k} n_k \in L$ and τ_{n_k} is primitive (an impossible plan) or SUPPORT $\langle Q_k@n_k\rangle$
      i. **Trace derivation:** see figure 6.3
      ii. **Resolve open condition:** see figure 6.4
2. **Termination:** see figure 6.1.
3. **Reduction:** Nondeterministically choose/add a new (instantiated from $\mathcal{M}$) method $[\tau_n \Rightarrow ctn_n]$ to T_*, add B_n to B, and add $O_n^{\emptyset}$ to O. Fail if B or O become inconsistent; otherwise remove all elements from L and D_* that mention τ_n. Respectively add $\langle \ell@n_i\rangle$ to *agenda* or $\langle n_i|\ell\rangle$ to D_* for each $n \xrightarrow{\ell} n_i$ or $n_i \xrightarrow{\ell} n$ removed.
4. **Recursive call:** Call UCREDUCE$(\langle T, B, O, L\rangle, \langle T_*, S_*, D_*\rangle, agenda)$

Figure 6.6: Structuring common steps into UCREDUCE

Lemma 17 *A partial plan's goal and task orientation are loop invariants during criticism.*

Proof: This proof has two cases depending on whether or not the open condition selected during a loop iteration is a non-literal. If it is a non-literal, lemmas 6 and 7 respectively imply that goal and task orientation are preserved .

In the case where the open condition selected is a literal, lemmas 9 and 12 imply that goal orientation is preserved because neither tracing the derivation nor resolving the open condition violate it. While 9 also implies that tracing does not violate task orientation, lemmas 11 and 13 together imply that resolving an open condition after tracing a derivation preserves task orientation. Therefore both goal and task orientation are preserved for both selected literals an selected non-literals. □

Lemma 18 *A partial plan's goal and task orientation are recursive invariants for* UCREDUCE.

Proof: Given that both goal and task orientation are loop invariants during criticism. The fact that the termination step leaves each item untouched implies that we need only consider the reduction step.

With respect to goal orientation, consider the links removed when reducing τ_n. For each link $n \xrightarrow{\ell} m$ removed, $\langle \ell @ m \rangle$ is added to *agenda* and $assured_m$ remains unchanged for any task τ_m. Since τ_n is abstract, none of these removals affect any set $preserved_m$. Also, adding $\langle m|\ell \rangle$ to D_* for each $m \xrightarrow{\ell} n$ removed implies that $required_m$ remains unchanged. Since each task's $required_m$ and $assured_m$ is unchanged, goal orientation is preserved.

To prove continued task orientation, consider its 5 conditions; one involves justification, and the other four involve T, B, O, and L. Justification requires that τ_m either appears in D_* or supports a link in L^θ. If τ_m cannot support a link in L^θ due to dropping a link in L, the addition of $\langle m|\ell \rangle$ to D_* preserves the condition.

The other four conditions involve matching T, B, O, and L with T^θ, B^θ, O^θ, and L^θ respectively for any θ satisfying B. The change to T_* implies that $\tau_n\backslash\theta$ and the links referring to it cannot appear in T^θ and L^θ respectively. Given that $\mathcal{G}^*$ uses a subset operation to compute L^θ_m and T^θ_m, dropping elements from L or T does not affect the task orientation conditions for these sets.

Finally the last two task orientation conditions, that B and O are supersets of O^θ and B^θ, are preserved while adding a method to T_* because all of the binding and ordering constraints are extracted from the method and explicitly inserted into B and O. $\Box$

Theorem 3 (Soundness) *Given* $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$ *any action sequence returned by the initial call to* UCREDUCE *solves* $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$.

Proof: Given that lemmas 1 and 2 imply that goal and task orientation hold for the initial call, an induction argument using the recursive invariants imply that these properties hold both for every call. With this result, lemma 17 implies that they both hold whenever the termination step executes; now consider when this step calls `sol()`.

Since T has no abstract tasks, the task network is primitive. Since D_* is empty, and *agenda* has no selectable tasks, *agenda* must also be empty. At this point, these properties combine with lemma 4 to imply that `sol()` returns a solution to the problem whenever UCREDUCE succeeds. $\Box$

Not surprisingly, our completeness proof is similar to that of UCPOP. The only difference is its explicit reliance on the soundness of ADL actions modeling tasks. The completeness proof for UCPARSE did not rely on this property because UCPARSE never generated or resolved an abstract task's precondition.

Theorem 4 (Completeness) *Let* $\langle \mathcal{I}, \mathcal{O}, \mathcal{M}, \rho \rangle$ *be a planning problem with sound ADL operators modeling abstract tasks and a solution sequence* $\{a_i\}_{i=0}^m$. *The initial call* *to* UCREDUCE *will return the sequence given an appropriate set of nondeterministic choices.*

Proof: Consider a solution sequence $\{a_i\}_{i=0}^{m}$, from lemma 3 we know that the sequence is consistent with the initial call's partial plan.

Now consider an iteration of the critic loop; it resolves either a literal or a nonliteral open condition. While lemma 8 directly implies that consistency can be preserved for nonliterals, lemmas 10 and 14 combine to imply the same for literals. Therefore consistency can be preserved while iterating through the critic loop until no selectable open conditions remain. Since there can only be a finite number of causal links in any given abstraction of the final solution plan, this loop is guaranteed to terminate.

Given that the sequence is consistent after a recursive call's critic loop, lemma 5 implies that either planning can continue to the next recursive call, or that SOL can take the partial plan and return the solution sequence. If planning can continue to the next recursive call, the reduction step decomposes an abstract task with a nondeterministically chosen decomposition method; thus the reduction step can maintain consistency.

Finally, each call to UCREDUCE either terminates or decomposes an abstract task. Since a solution sequence's decomposition of τ_{s} can only have a finite number of methods, lemma 5 implies that UCREDUCE must eventually terminate by calling SOL to return the final solution. □

6.4 Summary

This chapter has formally proved that both UCPARSE and UCREDUCE are sound and complete. Instead of proving these properties for each algorithm separately, we identified several contiguous sequences of one or more steps that are common to both algorithms. We proved properties of these sequences and combined these properties to prove soundness and completeness of our algorithms.

These proofs are based on induction arguments. For soundness we prove recursive invariants regarding attaining goals and performing tasks; these combine to imply that our algorithms can only return solutions. For completeness we prove a recursive invariant that each algorithm can maintain the consistency of its partial plan with a given solution while making progress toward producing the solution. This invariant combined with the finite size of any given solution implies that each algorithm, with the appropriate nondeterministic choices, is guaranteed to terminate and return the given solution.

BIBLIOGRAPHY

[Amarel 68] S. Amarel. On representations of problems of reasoning about actions. In D. Michie, editor, *Machine Intelligence 3*, pages 131–171, 1968.

[Barrett & Weld 94] A. Barrett and D. Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994.

[Barrett et al. 93] A. Barrett, K. Golden, J. Penberthy, and D. Weld. UCPOP user's manual, (version 2.0). Technical Report 93-09-06, University of Washington, Department of Computer Science and Engineering, September 1993. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[Brotsky 84] D. Brotsky. An algorithm for parsing flow graphs. AI-TR-704, MIT AI Lab, March 1984.

[Chapman 87] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[Currie & Tate 91] K. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, November 1991.

[Davis 90] E. Davis. *Representations of Commonsense Knowledge.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[Drummond 94] M. Drummond. Precondition achievement planning: framework *sans* application? In *Proceedings of the AAAI workshop on Comparative Analysis of AI Planning Systems*, July 1994.

[Erol et al. 94a] K. Erol, J. Hendler, and D. Nau. HTN planning: Complexity and expressivity. In *Proc. 12th Nat. Conf. on A.I.*, pages 1123–1128, July 1994.

[Erol et al. 94b] K. Erol, J. Hendler, and D. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proc. 2nd Int. Conf. on A.I. Planning Systems*, pages 249–254, June 1994.

[Etzioni 92] O. Etzioni. An asymptotic analysis of speedup learning. In *Proceedings of the Ninth International Conference on Machine Learning.* Morgan Kaufmann, July 1992.

[Fikes & Nilsson 71] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4), 1971.

[Garey & Johnson 79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[Kambhampati & Hendler 92] S. Kambhampati and J. Hendler. A validation structure based theory of plan modification and reuse. *Artificial Intelligence*, 55:193–258, 1992.

[Kambhampati 94] S. Kambhampati. Comparing partial order planning and task reduction planning: A preliminary report. Technical Report CS TR-94-001, Dept. of Computer Science and Engg, Arizona State University, March 1994.

[Kambhampati et al. 94] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. Department of Computer Science and Engineering TR-94-002, Arizona State University, 1994. To appear in Artificial Intelligence Special Issue on Planning and Scheduling.

[Knoblock 90] C. Knoblock. Learning abstraction hierarchies for problem solving. In *Proc. 8th Nat. Conf. on A.I.*, pages 923–928, August 1990.

[Korf 87] R. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, September 1987.

[Lan 95] *Scope and Abstraction: Two Criteria for Localized Planning*, 1995.

[McAllester & Rosenblitt 91] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. 9th Nat. Conf. on A.I.*, pages 634–639, July 1991.

[McDermott 91] D. McDermott. Regression planning. *International Journal of Intelligent Systems*, 6:357–416, 1991.

[Minton 90] S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2-3), March 1990.

[Minton et al. 89] S. Minton, C. Knoblock, D. Koukka, Y. Gil, R. Joseph, and J. Carbonell. PRODIGY 2.0: The Manual and Tutorial. CMU-CS-89-146, Carnegie-Mellon University, May 1989.

[Pednault 86a] E. Pednault. Formulating multiagent, dynamic-world problems in the classical planning framework. In M. Georgeff and A. Lansky, editors, *Reasoning about Actions and Plans*. Morgan Kaufmann, 1986. Proceedings of the 1986 Timberline Workshop.

[Pednault 86b] E. Pednault. *Toward a Mathematical Theory of Plan Synthesis*. PhD dissertation, Stanford University, December 1986.

[Pednault 88] E. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4(4):356–372, 1988.

[Pednault 89] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[Pednault 90] E. Pednault. Formulating multiagent, dynamic-world problems in the classical planning framework. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 675–710. Morgan Kaufmann, San Mateo, CA, 1990.

[Pednault 91] E. Pednault. Generalizing nonlinear planning to handle complex goals and actions with context-dependent effects. In *Proc. 12th Int. Joint Conf. on A.I.*, July 1991.

[Penberthy & Weld 92] J. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 103–114, October 1992. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[Reiter 80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Russell & Norvig 95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., 1995.

[Sacerdoti 75] E. Sacerdoti. The nonlinear nature of plans. In *Proceedings of IJCAI-75*, pages 206–214, 1975.

[Smith & Peot 92] D. Smith and M. Peot. A critical look at knoblock's hierarchy mechanism. In *Proc. 1st Int. Conf. on A.I. Planning Systems*, pages 307–308, June 1992.

[Tate 77] A. Tate. Generating project networks. In *Proc. 5th Int. Joint Conf. on A.I.*, pages 888–893, 1977.

[Tate et al. 94] A. Tate, B. Drabble, and J. Dalton. The use of condition types to restrict search in an ai planner. In *Proc. 12th Nat. Conf. on A.I.*, pages 1129–1134, 1994.

[Warren 74] D. Warren. WARPLAN: A system for generating plans. Memo No. 76, Univerity of Edinburgh, Department of Computational Logic, 1974.

[Wilkins 88] D. Wilkins. Causal reasoning in planning. *Computational Intelligence*, 4(4):373–380, 1988.

[Wilkins 90] D. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, November 1990.

[Wilkins 92] D. Wilkins. *Using the SIPE-2 Planning System, A Manual for SIPE-2 Version 4*. SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, 1992.

[Yang & Tenenberg 90] Q. Yang and J. Tenenberg. ABTWEAK: Abstracting a nonlinear, least-commitment planner. In *Proc. 8th Nat. Conf. on A.I.*, pages 204–209, August 1990.

[Yang 90] Q. Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6(1):12–24, February 1990.

[Young et al. 94] R. Young, M. Pollack, and J. Moore. Decomposition and causality in partial-order planning. In *Proc. 2nd Int. Conf. on A.I. Planning Systems*, pages 188–193, June 1994.