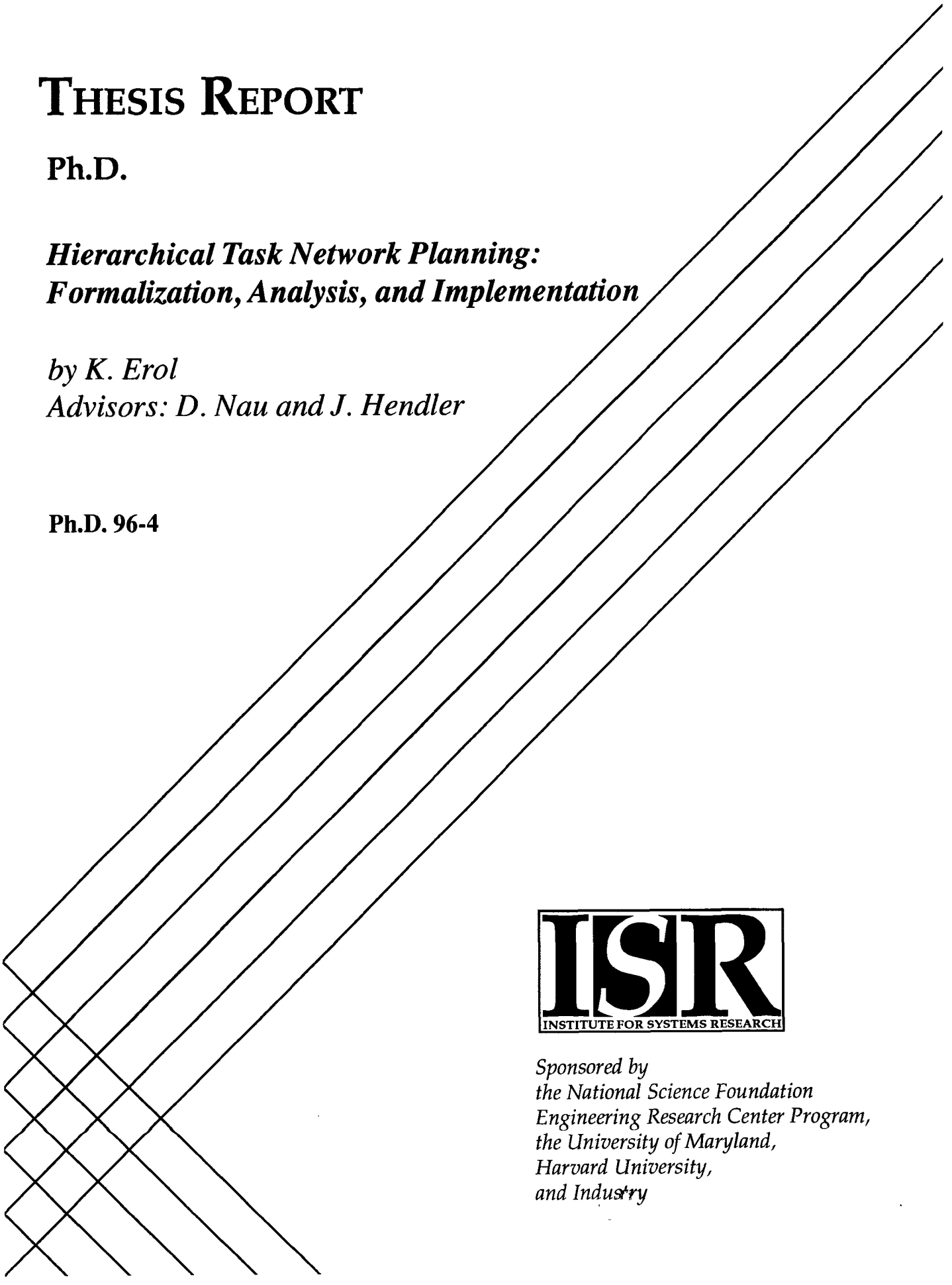
THESIS REPORT

Ph.D.

Hierarchical Task Network Planning: Formalization, Analysis, and Implementation

by K. Erol
Advisors: D. Nau and J. Hendler

Ph.D. 96-4



Sponsored by
the National Science Foundation
Engineering Research Center Program,
the University of Maryland,
Harvard University,
and Industry

Hierarchical Task Network Planning: Formalization, Analysis, and Implementation

Kutluhan Erol

Advisors
Drs. Dana Nau and James Hendler

University of Maryland
College Park, MD 20742

Abstract

Planning is a central activity in many areas including robotics, manufacturing, space mission sequencing, and logistics. As the size and complexity of planning problems grow, there is great economic pressure to automate this process in order to reduce the cost of planning effort, and to improve the quality of produced plans.

AI planning research has focused on general-purpose planning systems which can process the specifications of an application domain and generate solutions to planning problems in that domain. Unfortunately, there is a big gap between theoretical and application oriented work in AI planning. The theoretical work has been mostly based on state-based planning, which has limited practical applications. The application-oriented work has been based on hierarchical task network (HTN) planning, which lacks a theoretical foundation. As a result, in spite of many years of research, building planning applications remains a formidable task.

The goal of this dissertation is to facilitate building reliable and effective planning applications. The methodology includes design of a mathematical framework for HTN planning, analysis of this framework, development of provably correct algorithms based on this analysis, and the implementation of these algorithms for further evaluation and exploration. The representation, analyses, and algorithms described in this thesis will make it easier to apply HTN planning techniques effectively and correctly to planning applications. The precise and mathematical nature of the descriptions will also help teaching about HTN planning, will clarify misconceptions in the literature, and will stimulate further research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Planning involves reasoning about actions in order to find a way of achieving desired behaviors. It is an essential component of human intelligence, and it has a wide range of applications in areas such as robotics, manufacturing, space mission sequencing, and logistics. The size and complexity of these applications are rapidly growing beyond the capabilities of the people responsible for planning. There is great economic pressure to automate this process in order to reduce the cost of planning effort, and to improve the quality of produced plans.

Building one planning system for each application is not cost effective. Such a planning system is of little use beyond its specific application domain. It is very difficult to maintain and modify such a planning system as the requirements of the application change. Furthermore, building a reliable planning system is very demanding. Conventional programming languages such as C or Pascal do not provide any support for implementing complex planning applications.

Planning applications, although diverse in nature, share many common aspects. Thus, it is feasible to build general-purpose planning systems which can process the specifications of an application domain and generate solutions to planning problems in that domain. This avenue of research is called *domain-independent planning.*

Figure 1.1 shows a general view of domain-independent planning. Application experts prepare a specification of the application domain using a special planning language. Users of the planning system prepare specifications of planning problems that need to be solved. A general-purpose planning system, built by planning researchers, processes the specifications and produces solutions to those problems.

Building domain-independent planning systems is very challenging. Such a system must be practical. It must produce high quality solutions in a timely fashion, and it must meet the demands of planning applications. Domain independent planning systems must also have solid theoretical foundations. The language used for writing domain and problem specifications must have a precise syntax and semantics. Without precise semantics, it is very difficult, if not impossible, to develop correct and efficient algorithms for solving

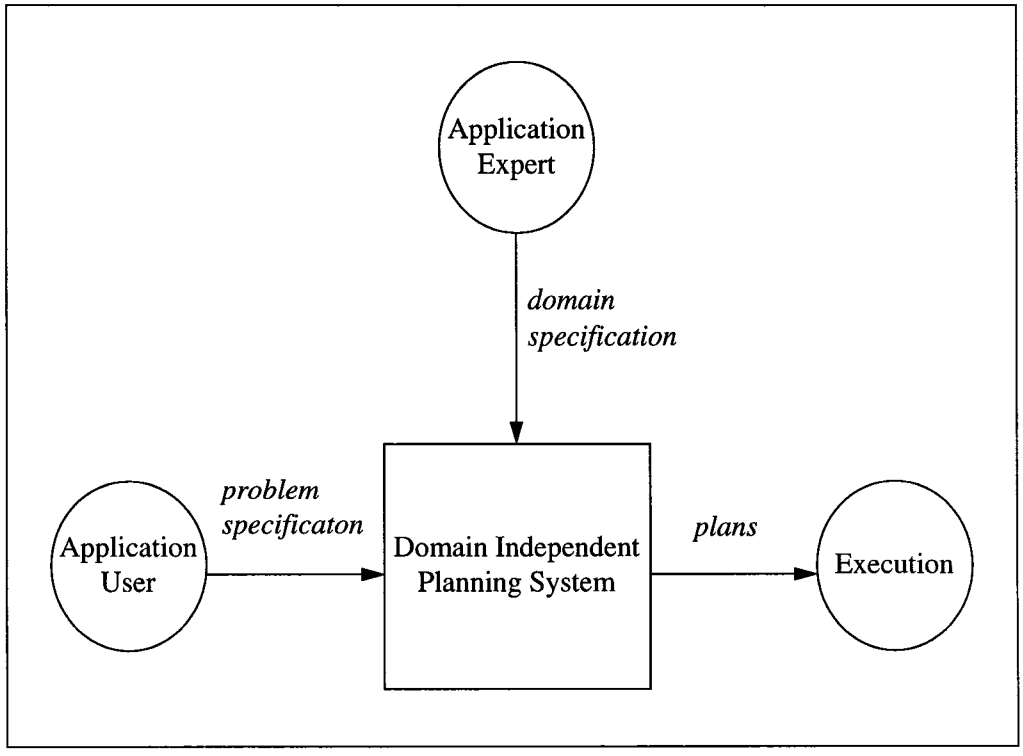


Figure 1.1: A General View of Domain-independent Planning

problems written in that language. A precise syntax and semantics also facilitate writing correct domain specifications. Application experts and users only need to learn the syntax and semantics of the language without having to learn the implementation details of complex planning systems. Furthermore, the language must be sufficiently expressive to represent the domain information necessary to solve problems in a reasonably wide range of applications.

Unfortunately, the interaction between the theoretical work and the more pragmatic, application oriented work on AI planning has been limited.

The theoretical work on domain-independent planning has been mostly on state-based planning[1] that stemmed from STRIPS [24] planning system, with roots in situation calculus [47]. The state-based planning paradigm has been deeply investigated. The representation (i.e., the STRIPS language) has been precisely defined, and the flaws in its semantics have been corrected [47, 45]. Many algorithms for finding plans have been developed and investigated [60, 15, 46, 10, 41, 54]. The computational complexity of solving planning problems represented in the STRIPS language has been studied in detail [12, 15, 22].

In spite of all this effort, state-based planning has been used mostly in the realm of simple "toy" domains, which were designed for research purposes only. The limitations of the type of information about an application domain that can be encoded in the STRIPS language, the computational cost of finding solutions, and the inadequacy of the mechanisms designed for building more efficient state-based planning systems prevent the transition to real world applications.

The more pragmatic work on domain-independent planning has paid less attention to formal foundations, and instead focused directly on building planning systems. Out of this research avenue was born a number of planning systems such as HACKER [63], NOAH [59], and NONLIN [64]. These systems established a new planning paradigm called *hierarchical task network(HTN) planning.*

HTN planning introduced many powerful ideas such as tasks and task decomposition, partial order planning, and least commitment. HTN planning systems showed much promise in building planning applications. For example, DEVISER [69] was used to develop a prototype for Voyager spacecraft mission sequencing, and SIPE [72] was used to build a prototype system for factory automation.

Unfortunately, HTN planning systems could not be developed to the level of successful software products for planning applications. I contend this is due to the lack of an underlying theoretical foundation. HTN planning techniques are heuristic in nature, and rather vaguely specified. Descriptions of these techniques are buried deep down in the implementation details of planning systems. As a result, HTN techniques are understood very little, and they cannot be reliably used to build planning applications. HTN planning techniques are powerful, but they still need to be improved and extended in order to make them work effectively in a wide range of applications. The lack of a precise understanding of HTN plan-

[1] As explained in Section 2.2.1, the terms "state-based planning" and "STRIPS-style planning" are interchangeably used to refer to planning techniques and algorithms which represent change in the world with STRIPS operators, and which represent desired behaviors as attainment goals in the final state. Examples are TWEAK, SNLP, and WARPLAN.

ning constructs impedes further research for improving them. Some of the ideas introduced in HTN planning have been adapted to state-based planning and investigated within that paradigm. However, the core ideas such as task networks and task decomposition, which are largely responsible for the success of HTN planning systems over state based planning systems, have been ignored as "efficiency hacks".

1.2 Research Issues

As the complexity of planning applications increases, and the correctness and quality of produced plans become more and more critical, there is a greater need for using formal methods to develop and analyze planning systems. In spite of three decades of research, building planning applications remains to be a formidable task. HTN planning techniques show promise in terms of flexibility of representation and computational performance, however there are a number of problems that need to be addressed:

- Existing HTN languages do not have a precise semantics. As a result, writing specifications for a planning problem using HTN constructs is so difficult that it is considered an art rather than a science. Even when a specification is provided, there is no criteria for verifying that it correctly represents the intended planning domain.
- HTN planning systems are not very reliable. They do not provide any guarantees that the solutions they produce are correct, or that they can always find the solutions. Furthermore, without a precise semantics that provides the definition for the set of solutions to a given HTN problem, it is not possible to develop correct HTN planning algorithms. Neither it is possible to evaluate existing algorithms for correctness.
- HTN planning systems usually find plans faster than state-based planning systems. Nonetheless, the speed of HTN techniques must be significantly improved to work well with real-world applications.

1.3 Approach

The focus of this dissertation is to correctly define, analyze, and explicate features of the design of HTN planning systems. The goal of this research effort is to support the development of efficient and correct HTN planning algorithms and to reduce the domain engineering effort in correctly specifying application domains using HTN constructs. The approach for reaching this goal has involved three stages: formalization, analysis and implementation.

1.3.1 Formalization

Formalization involves defining an HTN language together with the associated syntax and semantics, as presented in Chapter 3. The semantics provide precise definitions of the meanings of HTN constructs, and facilitates their correct use in writing domain specifications. The

semantics also provide a precise definition for the set of solutions to a given planning problem. This definition provides a criterion for developing correct HTN planning algorithms. Chapter 3 contains the first provably correct HTN planning algorithm UMCP (Universal Method Composition Planner).

This formal approach enforces the separation of the representation for planning problems from the implementation details of particular planning systems that can process the representation. It gives guidelines for planning researchers in developing correct HTN planning systems, and it reduces the domain engineering effort for preparing domain specifications by isolating it from the idiosyncrasies of implementation. A mathematical model of HTN planning also makes it amenable to the application of analytical tools.

1.3.2 Analysis

Analytical analyses can be very helpful in guiding the research efforts for finding efficient planning algorithms. They bring a deeper understanding of the domain specification and plan generation processes, and uncover the factors that influence the performance of HTN planning systems. They also identify the bottlenecks and computational difficulties involved in HTN planning.

Chapter 4 contains a complexity analysis which determines the computational cost of solving HTN problems under various restrictions. This analysis reveals which aspects of the HTN problems are difficult to handle, and where further research must be directed.

Analysis can also be helpful in discovering the similarities and differences among different planning paradigms. A better understanding of the relationship between two planning paradigms can facilitate adaptation of the results and techniques developed in one paradigm to the other.

Chapter 5 contains several definitions of expressivity for planning languages, extending the previous work on knowledge representation languages [5]. Based on these definitions, the HTN language can be evaluated in terms of expressive power in comparison to the STRIPS language, which is the de facto standard state-based planning language. This work proves that the HTN language is strictly more expressive than the STRIPS language. It also provides two polynomial transformations from the STRIPS language to the HTN language. These transformations are used to transfer several results developed in state-based planning to HTN Planning.

1.3.3 Implementation

The analysis provides insights to the design of correct and efficient HTN planning algorithms. Chapter 3 contains a provably correct but rather abstract HTN planning algorithm called UMCP. Based on the insights from the analysis results, correct and efficient algorithms are designed for each step of the abstract UMCP algorithm. As presented in Chapter 6, these algorithms have been implemented in the UMCP planning system for further exploration. The UMCP system serves as a testbed for experimenting with search control and commitment

strategies in order to improve the performance of HTN planning systems.

1.4 Organization

The layout of this dissertation reflects the steps in the research approach. Chapter 2 examines the issues introduced in Section 1.2 and presents a historical perspective to planning. Chapter 3 contains the definition of the HTN language and the associated semantics. This chapter also presents the abstract UMCP algorithm, and its correctness proof. The complexity analysis of HTN planning is discussed in Chapter 4. Chapter 5 presents several definitions for the expressivity of planning languages. Using these definitions and the complexity results from previous chapter, it investigates the similarities and differences between state-based planning and HTN planning. Chapter 6 describes further refinements of the UMCP algorithm, and present the implementation features of the resulting software architecture. Chapter 7 explains how to write specifications for a planning domain using the HTN language. It contains several sample problem specifications, and descriptions of how UMCP solves those problems. Finally, Chapter 8 provides a global perspective to the work contained in this dissertation, and points to future research directions for extending and improving the HTN paradigm.

Chapter 2

Related Work

This chapter presents a historical perspective to AI planning literature in order to provide a context within which the contributions of this dissertation can be evaluated. It is by no means intended as a comprehensive review of the literature, which can be found in the collection of papers edited by Allen, Hendler, and Tate [2].

The first section of this chapter describes the roots of classical AI planning. The second section contains a brief overview of some of the planning systems that evolved from these roots. Section 3 discusses the theoretical work on planning. This section also presents the attempts to formalize the ideas that emerged from implementations of planning systems. Finally, Section 4 discusses where the work described in this dissertation fits in, and how it relates to the recent and current research efforts.

2.1 The Early Years of AI Planning

The roots of classical AI planning lie in the discovery that first-order logic could be used to represent change in the world, and hence automated first-order theorem provers could be directly employed to solve planning problems. This discovery led to the development of *situation calculus* [47]. The details of situation calculus can be found in most introductory AI textbooks such as [52], but the general idea can be summarized as follows: atoms such as $p(x)$ that are normally used to represent static facts about the world are augmented by an extra term as in $p(x, s)$ to state that the predicate p is true of term x in situation s. Actions are represented using function symbols in the following manner: let the function symbol f_a denote an action a. The situation resulting from executing action a in situation s is represented by $f_a(s)$. The effects of executing an action are described by axioms. These axioms usually read as "if these conditions are true in a situation s, then those conditions will be true in the situation resulting from the execution of action a in s." Desired behaviors are represented as *attainment goals*, which are conditions (i.e., atoms) that are desired to be true in the world.

The planning problems can be posed as

"Does there exist a situation which satisfies the goal conditions and which is also reachable from the initial situation?"

Situation calculus allows the representation of context-dependent effects of actions. That is, an action can have different effects depending on the situation it is executed in. In return for this flexibility, situation calculus requires explicit axiomatization of all the conditions that are not affected by each action, in addition to the conditions that are affected by it. The axioms which describe the conditions that are *not* affected by a given action are called *frame axioms.* Frame axioms are usually of the form

"Condition c will be true in the situation resulting from executing action a at situation s, if c is already true in s, and action a does not make c false."

Situation calculus requires a frame axiom for each action – condition pair. This gives rise to the *frame problem*: huge numbers of axioms are necessary for describing even simple toy planning domains. Another disadvantage of situation calculus is performance related: general purpose theorem provers cannot exploit the structure of planning problems and thus they are horrendously slow in solving planning problems.

Both the representation and the performance issues were addressed in the STRIPS planning system [24]. The frame problem was dealt with by the STRIPS assumption:

Executing an action does not change the truth value of a condition unless the condition is one of the effects of the action.

This assumption is built into the STRIPS system, and hence, for each action only the conditions it changes need to be specified. Each action is described as a STRIPS operator, which consists of a precondition list, an add list, and a delete list. The precondition list specifies the conditions under which the action is executable. The add list and delete list, respectively, specify the conditions that are going to become true and those that are going to become false in the state resulting from the action execution. Context-dependent effects are not allowed. The syntax used for encoding operators, and their associated semantics will be referred to as the STRIPS language.

The STRIPS algorithm uses precondition chaining for finding solutions to planning problems. STRIPS maintains a goal stack, which initially contains the goals in the planning problem. At each iteration, STRIPS examines the goals on top of the stack, and chooses an operator which has effects that match one or more goal conditions that are not true in the current state. If the preconditions of the operator are satisfied in the current state, the operator is applied to compute a new current state. Otherwise, the operator is added to the working plan, and its preconditions are pushed onto the goal stack as new subgoals. Whenever no operator has a matching effect with the goals on top of the goal stack, the algorithm backtracks. The STRIPS algorithm is heavily influenced by GPS [51] (General Problem Solver), whose operator choice is driven by minimizing the difference between the current state and the goal state.

The STRIPS algorithm deals with conjunctive goals of the form $G_1 \wedge G_2$ as follows: First find a plan that achieves G_1 from the initial state, and then find a plan that achieves G_2 from the final state of the previous plan. If it happens that the second plan undid the first goal, STRIPS tries to accomplish G_2 first, and then G_1. This procedure works well as long as there are no interactions among the subplans for the goals. However, even very simple planning domains contain *nonserializable goals*. These are goals for which there is no solution unless the plans for each goal are interleaved. The STRIPS system ignores nonserializable goals by making the *linearity assumption* [63]:

> Subgoals are independent and thus can be sequentially achieved in an arbitrary order.

The linearity assumption significantly simplifies devising algorithms for dealing with conjunctive goals. While constructing a solution, such algorithms keep the plan steps (actions) totally ordered, and new steps are added only to the beginning or to the end of the plan. However, planning algorithms based on the linearity assumption may fail to find a solution, even though there exists a solution. Thus they are not *complete*. A complete algorithm always finds a solution, whenever there exists a solution.

The search space for the STRIPS language is large, because the number of possible states is exponential in the number of atoms in the language. Even with the linearity assumption, the STRIPS system could solve only very small planning problems in any reasonable amount of time.

2.2 The Planning Systems in 1970's

The development of the STRIPS system initiated a huge body of research. In the following years many new planning systems were built. These systems tried to address the shortcomings of the STRIPS system. Some of them focused on improving the performance, and some of them focused on expanding the set of planning problems that can be correctly dealt with. Embedded in those systems were many powerful ideas, some of which are listed below:

1. Abstraction hierarchies for improving performance,
2. Relaxation of the linearity assumption for dealing with nonserializable goals,
3. Task networks and task decomposition, for representing and solving complex problems,
4. Partial-order planning for improving performance and dealing with nonserializable goals,
5. Least commitment strategies for improving performance,
6. Filter conditions for reducing the size of the search space,
7. "Critics" mechanism for dealing with interactions.

All of these planning techniques will be discussed in detail, following some clarifications in the planning terminology.

2.2.1 Planning Terminology

There is some confusion in planning terminology, perhaps because several planning systems have introduced multiple techniques to the planning field simultaneously. In this section, I will define the meanings of several terms as they are used in this dissertation to prevent misunderstandings.

Most planning systems work iteratively, incrementally constructing the final plan. The incomplete plan at each iteration is called the *working plan.* Initially the working plan is usually empty. Some planning systems, for convenience, choose to represent the initial state and the goals as two special steps in the initial working plan.

A planning system that keeps the steps in the working plan totally ordered is called a *total-order* planner. A planning system that keeps the steps in the working plan partially-ordered is called a *partial-order planner.*

Any planning system that makes the linearity assumption is called a *linear planner.* A planning system that does not rely on that assumption is called a *nonlinear planner.* The linearity assumption was defined in Section 2.1.

Planning systems that use the STRIPS operators for representing actions and solve problems which involve finding a path from initial states to goal states are called *state-based*, or *STRIPS-style* planners. It does not matter which techniques are used (total-order, partial order, regression, etc.) for producing the plan. Thus according to this definition, STRIPS [24], SNLP [46], and TWEAK [15] are state-based planners.

A planning system that represents planning problems as task networks, and uses task decomposition to construct the final plan is called a *hierarchical task network (HTN) planner.*

Sometimes these terms are used with different meanings in the literature. The terms "state-based" and "linear" are often used to refer to total-order planners. The term "nonlinear" is sometimes used to refer to partial-order or HTN planners. This type of usage is misleading and thus it is avoided in this dissertation.

2.2.2 Abstraction Hierarchies

Abstraction hierarchies were introduced by Sacerdoti in 1974, in the ABSTRIPS system [60]. Their purpose is to cope with the huge size of the search space and to reduce planning time by focusing the planning effort on the most critical parts of the problem first. Each condition (i.e., predicate or proposition) in the planning domain is assigned a criticality level using heuristic methods. Planning proceeds in levels, starting with the highest criticality level. At each level, all the conditions with lower criticality levels are ignored as details. The plan found in one level is used as a skeleton in finding a plan in the next lower level iteratively, until the bottom level is reached, which produces the final plan.

Abstraction hierarchies are given that name because of the abstraction resulting from ignoring some conditions, and the hierarchy consisting of the criticality levels. However, like the variable-ordering heuristics found in the constraint satisfaction literature [25, 50], abstraction hierarchies can also be perceived as a means of deciding which part of the problem to work on next.

2.2.3 Relaxation of the Linearity Assumption

Planning algorithms based on the linearity assumption could not solve problems containing nonserializable goals. As discussed earlier, subplans for nonserializable goals need to be interleaved. The planning systems of 1970's introduced several techniques for dealing with nonserializable goals.

The HACKER [63] system was developed by Sussman in 1975 for manipulating a robot arm. HACKER deals with interacting subgoals using the *critics* mechanism. This procedural mechanism identifies and patches conflicts based on heuristic methods. HACKER stores patches that worked well, to be used later in similar problems. Thus it can be considered as the ancestor of case-based planning systems [42, 40]. A case-based planner stores solutions to previous planning problems in a case-base, and solves each new planning problem by adapting the solution of a similar old problem from the case-base.

Another way of dealing with nonserializable goals is to keep the plan steps totally ordered, but to allow new steps to be inserted at any point in the plan. Warren [71] in 1974 and Waldinger [70] in 1977 took this approach and used goal regression and action regression techniques, respectively, for dealing with conjunctive goals. Their planning systems would find a plan for the first subgoal, and then modify that plan by inserting new steps as necessary to achieve the next subgoal without undoing the previously achieved subgoals.

HTN planning systems such as NOAH [59] and NONLIN [64] introduced *partial-order planning*, which facilitates handling nonserializable goals. In partial order planning, the decision regarding where a new step should be added in the working plan is deferred. Plan steps are pairwise ordered only to the extend necessary to resolve conflicts. Thus, subplans for goals can be freely interleaved.

Partial order planning requires significantly more complicated algorithms than total order planning. The intermediate states in a plan cannot be constructed if the steps are not totally ordered, and thus it becomes difficult to assess whether a condition is going to be true before a given plan step. On the other hand, partial-order planning can provide gains in efficiency, because the planner may not need to consider all possible orderings. The trade-off between total-order versus partial-order planning has been studied extensively in the last several years, albeit in the framework of state-based planning, instead of HTN planning. These studies will be discussed in Section 2.3.5.

Planning systems that do not depend on the linearity assumption are called *nonlinear planners*. Since most of the nonlinear planners have also been partial-order planners, the term "nonlinear" is often incorrectly used to refer to partial-order planners, and sometimes to refer to HTN planners.

2.2.4 Least Commitment

Planning systems, while refining a partial solution into a final plan, make several types of decisions: which actions to choose, how to order the steps in the plan, and how to (or how not to) instantiate variables. The first generation of planning systems, usually made the decisions regarding ordering and variables before the choice of actions. Such a planning system would immediately order each new step inserted to the plan with respect to the existing steps, and instantiate all the variables in the new step. All possible orderings and variable instantiations would be explored in a depth-first manner. This results in a very large branching factor, and it is rather inefficient for many planning problems.

In response to this situation, HTN planners such as NOAH [59] typically work by introducing ordering and variable binding restrictions only when necessary to resolve conflicts. This kind of strategy has been called *least commitment*. Least commitment strategy may reduce amount of backtracking in a number of planning situations where a total-order planner would try and backtrack from many possible orderings.

The MOLGEN [62] system developed by Stefik in 1981 further enhanced least commitment for variable bindings. MOLGEN was designed to work in the area of experiment planning for molecular biology. The choice of objects in those experiments are rather important. MOLGEN uses constraint-based techniques to guide the search in finding the right variable instantiations for the objects.

The least-commitment strategies point to a more general problem of commitment: At each iteration, a planning system can make several types of commitment decisions, including variable bindings, orderings, and action selection. The order in which these commitments are made has a large impact on the efficiency of a planning system, and the question becomes how to choose the best type of commitment to make at each iteration. Any given commitment strategy may be efficient in some application domains, and slow in some others. The approach taken in this dissertation, as described in Section 6.3, is to provide a general commitment mechanism which can be customized according to the needs of each planning application.

2.2.5 Task Networks and Task Decomposition

In the early planning systems, the emphasis was on states. Actions were viewed only as a means for affecting state changes. HTN planning brought a very different perspective, placing the emphasis on the activities to be planned for (i.e. the tasks) and the interactions among them. Planning proceeded by decomposing each task into simpler tasks and resolving the conflicts among tasks iteratively, until a conflict-free plan consisting of primitive tasks could be found.

Task decomposition techniques were initially introduced in the NOAH [59] system developed by Sacerdoti in the mid-1970's, and were enhanced in NONLIN [64] system designed by Tate in 1977. NOAH represented planning problems as partially ordered lists of tasks (hence the term "task network"). The information on how to decompose tasks was encoded procedurally in the so called "soup code." NOAH used an improved version of critics, a heuristic procedural mechanism initially introduced by HACKER [63], to detect and resolve

interactions among tasks. Interactions were resolved mostly by placing ordering or variable binding restrictions.

NONLIN replaced the procedural soup code of NOAH with *opschemas*, which declaratively represented how to decompose tasks. NONLIN also introduced two data structures called the *TOME* (table of multiple effects) and the *GOST* (goal structure) for keeping track of which tasks have a given effect, and also which tasks can influence a given condition. It employed backtracking to recover from incorrect planning decisions. NOAH had no such mechanism, and it would report failure in those cases.

In the following years, the DEVISER [69] system developed by Vere in 1983 extended the NOAH and NONLIN systems to handle actions with durations and activities with temporal constraints.

The work on HTN planning was further advanced by SIPE [73], developed by Wilkins in 1983. SIPE emphasizes resource management and also the interactions with the user of the planning system. SIPE employs a taxonomy for storing the features of objects in the planning domain. The UMCP planning system, presented in Chapter 6, uses a similar technique for representing the initial state of a planning problem.

The O-PLAN [17] system designed in 1991 by Currie and Tate further extended the NONLIN framework to a more general plan creation and execution platform. O-PLAN represents plans using several elaborate types of constraints. O-PLAN has introduced the *triangle model of activity* to integrate automatic manipulation of plans with human interaction and information acquisition. Recently this system was enhanced further resulting in a system called O-PLAN2 [65].

Unlike state-based planners which rarely were used for practical applications, HTN planners met with considerable success in building applications. NOAH was applied to mechanical engineers apprentice supervision, NONLIN was applied to electricity turbine overhaul, DEVISER was applied to Voyager spacecraft mission sequencing, and SIPE was applied to aircraft carrier mission planning [2].

2.2.6 Filter Conditions

Filter conditions were introduced in NONLIN [64] as a mechanism to reduce the size of the search space by detecting and filtering out undesirable solutions in early stages of planning. For example, consider a medical domain. Suppose a particular drug is administered only to patients with high fever. We would want to eliminate those plans that attempt to increase the temperature of the patient so that the drug can be administered. Furthermore, we would like to save the time the planner spends in trying to establish the high fever predicate. Such a condition is represented as a filter condition, instead of a precondition. If the filter condition is not already true, NONLIN backtracks; it does not attempt to establish it.

Evaluating filter conditions is not a simple task, since in the early stages of planning, a task network might not contain enough detail to determine whether a filter condition will be true at a point in the task network. NONLIN took a conservative approach by evaluating each filter condition as soon as it was introduced in a task network, and pruning those task

networks whose filter conditions were not determined to be necessarily true. This approach reduces the search space considerably, but compromises completeness in situations where the filter conditions are possibly but not necessarily true. The UMCP planning system implemented as part of this dissertation takes a different approach that preserves completeness, as discussed in Chapter 6.3.

2.2.7 Critics

Another important aspect of planning that is not addressed in current analytic efforts is the general use of critics. Historically speaking, critics were introduced into NOAH [59] as a procedural, heuristic mechanism to identify and deal with several kinds of interactions (not just deleted preconditions) among tasks in a task network. After each task decomposition, a set of critics are consulted so as to recognize interactions and resolve conflicts. This approach provides a more general framework for detecting interactions than is available in most STRIPS-style planners. Based in part on Sussman's earlier work in HACKER [63], Sacerdoti [59] identified three critics of general use:

- *Resolve Conflicts.* The conflicts handled by this critic, later referred to as "deleted-condition" interactions, have received the bulk of the attention in the literature.
- *Use existing objects.* This critic dealt with resources, rather than temporal ordering relations.
- *Eliminate redundant preconditions.* This critic identified and utilized opportunities by recognizing cases where a goal task was achieved inadvertently while achieving another task. When such opportunities were recognized, this critic would convert the already achieved goal task into a "phantom" condition.

In addition to the interactions handled by these critics, a number of other situations that can arise in planning have been identified in the literature:

- For his DEVISER system, Vere [69] has discussed temporal interactions between the times at which actions must occur.[1] He has discussed temporal windowing and an analysis thereof to eliminate possible reductions.
- Wilkins' SIPE system [73] has added a number of different mechanisms for recognizing resource interactions and for allowing user preferences to be considered when making a choice between reductions.
- Yang, Nau, and Hendler [76] have introduced a general "action-precedence" interaction that can be exploited in some planning situations.

[1] See also [18].

- Nau and Gupta [28] have identified "enabling-condition" interactions as the culprit that makes finding optimal plans in the blocks-world domain to be NP-hard. Such interactions arise when a side-effect of achieving one goal is to make it easier to achieve another goal.

- To handle iteration in plans, Drummond [19] has proposed several extensions to the procedural net, and an extension to Sacerdoti's resolve-conflicts critic.

- A number of special-purpose "domain-dependent" planning systems have identified interactions occurring only in the particular domain for which the system is being developed. Typically, special-purpose heuristics are introduced to exploit this knowledge.

As can be seen, the many interactions which need to be handled during planning go beyond the (relatively) well-understood deleted-condition interaction. To handle these interactions, implemented planning systems usually use critics or similar mechanisms. However, these mechanisms are procedural and heuristic in nature. To reason about the properties of such mechanisms, a general model of interactions and critics is clearly needed. One important aspect of such a model is that it can help, in part, to unify "domain-independent" and "domain-dependent" planning systems: in many of the latter, the specialized knowledge can be modeled as using a large set of special-purpose critics. The formal framework presented in this dissertation in Chapter 3 provides just such a model. Chapter 6.3 presents a way of implementing domain-independent critics that preserves soundness, completeness, and systematicity, based on that framework.

2.3 Formalization and Analysis Efforts

The 1980's brought formal approaches to planning. The semantics of the STRIPS language was freed of flaws [45] and brought to its current standard form as presented in Section 2.3.1. A significant amount of research effort went into attempts at formalizing and analyzing the ideas buried inside the implementation details of the systems summarized in Section 2.2. Among those ideas, the abstraction hierarchies, the ways to relax the linearity assumption, partial and total-order planning have been studied quite well. There has been some work on filter conditions and commitment strategies, but the rest of these ideas have been studied very little and mostly dismissed as mere "efficiency hacks."

This section contains a brief overview of some of the work on formalization and analysis. Note that almost all of these studies have been done in the framework of state-based planning, although many of the ideas originated in the context of HTN planning. Carrying those studies back to the HTN framework is not a simple task, but it is significantly facilitated by the work presented in this dissertation.

2.3.1 The STRIPS Representation

Most of the state-based planning systems use the STRIPS representation. HTN planning systems also use variations of the STRIPS representation for describing the world and change in the world. Thus a good understanding of the STRIPS representation is very important.

The STRIPS representation has gone through several changes. In the original STRIPS planner [24], the planning operators' precondition lists, add lists, and delete lists were allowed to contain arbitrary well-formed formulas in first-order logic. However, there were a number of problems with this formulation, such as the difficulty of providing a well-defined semantics for it [45]. Thus, in subsequent work, researchers have placed several restrictions on the nature of the planning operators [52]. Typically, the precondition lists, add lists and delete lists contain only atoms, and the goal is a conjunct of ground or existentially quantified atoms. This representation has become the de facto standard STRIPS representation, used by most of the state-based planning systems [15, 46, 77]. The description of the STRIPS representation below, which I developed jointly with Dr. Nau and Dr. Subrahmanian [22], is in accordance with this formulation.

Definition 1 Let $\mathcal{L}$ be any first-order language generated by finitely many constant symbols, predicate symbols, and function symbols. Then a *state* is any finite set of ground atoms in $\mathcal{L}$.

Intuitively, a state tells us which ground atoms are currently true: if a ground atom A is in state S, then A is true in state S, and if $B \notin S$, then B is false in state S. Thus, a state is simply an Herbrand interpretation for the language $\mathcal{L}$, and hence each formula of first-order logic is either satisfied or not satisfied in S according to the usual first-order logic definition of satisfaction.

Definition 2 Let $\mathcal{L}$ be an ordinary first-order language. Then a *planning operator* α is a 4-tuple $< \text{Name}(\alpha), \text{Pre}(\alpha), \text{Add}(\alpha), \text{Del}(\alpha) >$, where

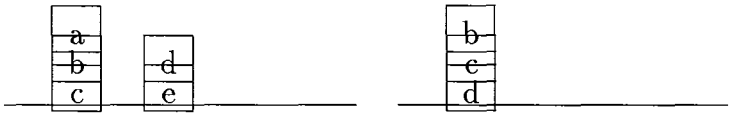
1. $\text{Name}(\alpha)$ is a syntactic expression of the form $\alpha(X_1, \ldots, X_n)$ where each X_i is a variable symbol of $\mathcal{L}$;

2. $\text{Pre}(\alpha)$ is a finite set of literals, called the *precondition list* of α, whose variables are all from the set $\{X_1, \ldots, X_n\}$;

3. $\text{Add}(\alpha)$ and $\text{Del}(\alpha)$ are both finite sets of atoms (possibly non-ground) whose variables are taken from the set $\{X_1, \ldots, X_n\}$. $\text{Add}(\alpha)$ is called the *add list* of α, and $\text{Del}(\alpha)$ is called the *delete list* of α.

Observe that negated atoms are allowed in the precondition list, but not in the add and delete lists.

When defining a planning operator α, often $\text{Name}(\alpha)$ will be clear from context. In such cases, the name will be omitted.



(a) initial configuration (b) goal configuration

Figure 2.1: A sample blocks world problem.

Definition 3 A *first-order planning domain* (or simply a *planning domain*) is a finite set of planning operators $\mathcal{O}$.

Definition 4 A *goal* is a conjunction of atoms which is existentially closed (i.e., the variables, if any, are existentially quantified).

Definition 5 A *planning problem instance* is a triple $\mathbf{P} = (S_0, \mathcal{O}, G)$, where S_0 is the initial state, $\mathcal{O}$ is a planning domain, and G is a goal.

Example 2.3.1 (Blocks World) Suppose we want to talk about a blocks-world planning domain in which there are five blocks a, b, c, d, e, along with the "stack", "unstack", "pickup", and "putdown" operators used by Nilsson [52]. Suppose the initial configuration is as shown in Fig. 2.1(a), and the goal is to have b on c on d, as shown in Fig. 2.1(b). Then the language, operators, planning domain, and planning problems are defined as follows:

1. The language $\mathcal{L}$ will contain five constant symbols a, b, c, d, e, each representing (intuitively) the five blocks. $\mathcal{L}$ will contain no function symbols, and will contain the following predicate symbols: "handempty" will be a propositional symbol (i.e. a 0-ary predicate symbol), "on" will be a binary predicate symbol, and "ontable", "clear", and "holding" will be unary predicate symbols. In addition, there will be a supply of variable symbols, say, $X_1, X_2, \ldots$. Note that operator names, such as "stack", "unstack", etc., are not part of the language $\mathcal{L}$.

2. The "unstack" operator will be the following 4-tuple:

$$\begin{aligned} \text{Name(unstack)} &= \text{unstack}(X_1, X_2) \\ \text{Pre(unstack)} &= \{\text{on}(X_1, X_2), \text{clear}(X_1), \text{handempty}()\} \\ \text{Del(unstack)} &= \{\text{on}(X_1, X_2), \text{clear}(X_1), \text{handempty}()\} \\ \text{Add(unstack)} &= \{\text{clear}(X_2), \text{holding}(X_1)\} \end{aligned}$$

The "stack", "pickup", and "putdown" operators are defined analogously.

3. The planning problem will be $(S_0, \mathcal{O}, G)$, where S_0, $\mathcal{O}$ and G are as follows:

$$\begin{aligned} S_0 &= \{\text{clear}(a), \text{on}(a,b), \text{on}(b,c), \text{ontable}(c), \text{clear}(d), \\ &\quad \text{on}(d,e), \text{ontable}(e), \text{handempty}()\}; \\ \mathcal{O} &= \{\text{stack}, \text{unstack}, \text{pickup}, \text{putdown}\} \\ G &= \{\text{on}(b,c), \text{on}(c,d)\}. \end{aligned}$$

Definition 6 Given an initial state S_0, and a set of operators $\mathcal{O}$, let α be an operator in $\mathcal{O}$ whose name is $\alpha(X_1, \ldots, X_n)$, and θ be a substitution that assigns ground terms to each $X_i, 1 \leq i \leq n$. Suppose that the following conditions hold for states S and S':

$$\begin{aligned} &\{A\theta : A \text{ is an atom in } \text{Pre}(\alpha)\} \subseteq S; \\ &\{B\theta : \neg B \text{ is a negated literal in } \text{Pre}(\alpha)\} \cap S = \emptyset; \\ &S' = (S - (\text{Del}(\alpha)\theta)) \cup (\text{Add}(\alpha)\theta)\,. \end{aligned}$$

Then it is said that α is *θ-executable* in state S, *resulting* in state S'. This is denoted symbolically as

$$S \stackrel{\alpha,\theta}{\Longrightarrow} S'.$$

Definition 7 Given a planning problem instance $\mathbf{P} = (S_0, \mathcal{O}, G)$, a *plan that solves* $\mathbf{P}$ is a sequence $\alpha_1\theta_1, \ldots, \alpha_n\theta_n$ of ground planning operators such that

$$S_0 \stackrel{\alpha_1,\theta_1}{\Longrightarrow} S_1 \stackrel{\alpha_2,\theta_2}{\Longrightarrow} S_2 \cdots \stackrel{\alpha_n,\theta_n}{\Longrightarrow} S_n \tag{2.1}$$

and G is satisfied by S_n, i.e. there exists a ground instance of G that is true in S_n. The *length* of the above plan is n.

Given a planning problem instance, one might be interested in *any* plan that solves this instance. Alternatively, one might be interested in the shortest plan that solves this instance. Both of these existence and optimality questions are formally defined below as decision problems:

Definition 8 PLAN EXISTENCE is the following problem:

> Given a planning problem instance $\mathbf{P} = (S_0, \mathcal{O}, G)$, does there exist a plan in $\mathbf{P}$ that achieves G?

Definition 9 PLAN LENGTH is the following problem:

> Given a planning problem instance $\mathbf{P} = (S_0, \mathcal{O}, G)$ and an integer k encoded in binary, does there exist a plan in $\mathbf{P}$ of length k or less that achieves G?

2.3.2 State-based Planning Algorithms

Once the representation issues in the STRIPS language were resolved, a number of provably correct planning algorithms were developed. The most influential algorithms among those have been TWEAK [15] by Chapman in 1987, SNLP [46] by McAllester in 1991, and UCPOP [54] by Penberthy and Weld in 1992. This section contains a brief overview of these planning algorithms.

2.3.2.1 TWEAK

TWEAK [15] is the first provably correct partial-order planning algorithm. It works with incomplete plans, which are iteratively refined to a complete solution. An incomplete plan in TWEAK consists of a set of plan steps. Each step is associated with an operator. Along with the incomplete plan, TWEAK maintains a partial order graph on the steps, and a list of noncodesignation constraints on the variables in the operators of the plan. Noncodesignation constraints are of the form $v \neq c$, or $v_1 \neq v_2$, stating that variable v cannot have value c, or variables v_1 and v_2 cannot have the same value, respectively.

At the heart of the TWEAK algorithm lies a definition called *the modal truth criterion* for evaluating a condition when the steps of the plan are only partially ordered. The modal truth criterion is defined as follows: [15]

> A proposition p is necessarily true in a situation s iff two conditions hold: there is a situation t equal or necessarily previous to s in which p is necessarily asserted; and for every step C possibly before s and every proposition q possibly codesignating with p which C denies, there is a step W necessarily between C and s which asserts r, a proposition such that r and p codesignate whenever p and q codesignate.

The modal truth criterion roughly has the following meaning: for a condition p to be true at a target situation s, there must be some step t before it that necessarily asserts it, and for any other step that may come between t and s and falsify the condition, there is a corresponding step W (referred to as a *white night*) that makes the condition true again.

Planning in TWEAK starts with a null incomplete plan that has an initial step that asserts the conditions true in the initial state, and a final step that has the plan goals as preconditions. At each iteration TWEAK evaluates each precondition of every step in the plan using the modal truth criterion. If all the preconditions are necessarily true, then any total ordering of the incomplete plan consistent with the partial order has a ground instance consistent with the noncodesignation constraints on the variables.[2] Each such ground instance is a solution to the planning problem. When TWEAK discovers that a precondition is not necessarily true, it tries to make that condition true by adding a new step to the plan

[2]In order to ensure that there always exist a ground instantiation that satisfies the noncodesignation constraints on variables, TWEAK assumes the application domain contains infinitely many constant symbols. This assumption is relaxed in subsequent algorithms.

or finding an already existing step which has the desired effect (corresponding to the step t in the modal truth criterion). The threats from any step C which can deny the precondition is resolved by placing ordering or variable binding restrictions.

In evaluating a condition, the modal truth criterion assumes that the preconditions of all the other steps are satisfied. This does not cause a problem for TWEAK since it checks all the preconditions before it halts with a solution. Kambhampati and Nau have investigated this issue in depth and provided alternative definitions to the modal truth criterion [38].

In many ways TWEAK was an attempt at formalizing the ideas in the planners discussed in Section 2.2. It was very successful at providing a formal framework for partial order planning, and stimulated a lot of further research. However, it left out several important ideas such as task networks and task decomposition, which are studied in this dissertation.

2.3.2.2 SNLP

SNLP(Systematic Nonlinear Planner) was developed by McAllester in 1990 [46], in an effort to design an efficient, provably correct planning algorithm. There was already a sound and complete planning algorithm TWEAK, described in the previous section; however, TWEAK performs rather slowly on many planning problems.

One reason TWEAK performs rather slowly is that it does not protect the conditions it has previously established. Not protecting those conditions saves the time in bookkeeping for recording and checking previously established conditions. On the other hand, each time TWEAK establishes a condition, there is a possibility it will undo a previously established condition, which will need to be reestablished. Reestablishing that condition can likewise undo some other previously established conditions.

As explained in Section 2.2, NONLIN employed specialized data-structures to record and protect previously established conditions. The SNLP algorithm provided a formal version of these, which have been called *causal links.*

The SNLP algorithm is fairly similar to the TWEAK algorithm. However, when a precondition p of a step s is established using a step t, SNLP records a causal link $< t, p, s >$, stating that p will remain true from the end of step t till the beginning of step s. Step t is called the *establisher* of condition p. All the steps in the plan are required not to *threaten* the causal link. That is, each step is ordered either outside the interval between t and s, or its effects are made to noncodesignate with $\neg p$ by adding noncodesignation constraints on the variables. Every time a new step is inserted into the working plan, the same procedure is applied to make sure the new step cannot threaten any of the causal links.

As a possible way for improving efficiency, SNLP introduced the concept of *systematicity*:

> The set of solutions for the incomplete plans at every branch of the search tree must be mutually disjoint.

Enforcing systematicity ensures each incomplete plan is investigated at most once, since there cannot be multiple paths leading to any incomplete plan in the search space. Enforcing systematicity in a partial planner is somewhat tricky, as there are many ways of obtaining

a given partial order. SNLP guarantees systematicity by utilizing *positive threats*: Given a causal link $< t, p, s >$, SNLP protects it not only from steps that may deny p, but it also protects it from other steps that may assert p. This results in a unique establisher for every condition, and the set of solutions at alternative branches of the search tree are disjoint, as the establishers will be different.

Initially it was believed that SNLP would always work faster than TWEAK, but problems where TWEAK performs better were identified later. In situations with multiple candidate steps that can serve as establishers, SNLP algorithm requires immediately committing to one of the candidates. This commitment is sometimes premature, and thus a wrong candidate may be selected which results in backtracking later on. SNLP also incurs significant overhead while protecting the causal links. Tradeoffs between systematicity versus commitment were investigated by Kambhampati in [36].

TWEAK and SNLP represent two opposite extremes in protecting none versus all of the established conditions. In the following years, intermediate techniques, which protect only some of the previously established conditions, were developed [56, 61, 34].

2.3.2.3 UCPOP

UCPOP [54] is another sound and complete planning algorithm. It was designed by Weld and Henks in 1992. Rather than the STRIPS language, UCPOP is based on the ADL language developed by Pednault [53] in his dissertation work. ADL is significantly more complex than the STRIPS language, but in order to facilitate an efficient implementation, it was considerably stripped down, and except for context-dependent effects (i.e., effects that may take place depending on the input situation) and quantification over variables, the underlying language of UCPOP is very similar to the STRIPS language.

The UCPOP algorithm is similar to the SNLP algorithm in the aspects of establishing conditions using causal links and threat removal. In order to handle the context-dependent effects and quantifiers, UCPOP uses *secondary preconditions*. Two types of secondary preconditions are causation and preservation conditions. Causation conditions of an action a with respect to a proposition p refer to the conditions that must be true before a is executed so that a will assert p. Preservation conditions refer to the conditions that must be true before a is executed so that a will not make p false. Basically, the causation conditions provide the information about the possible ways a step can establish a condition, and the preservation conditions provide the information about the possible ways threats from a given step can be removed.

2.3.2.4 A Generic Planning Algorithm

In addition to the planning algorithms described above, numerous other planning algorithms have been devised for state-based planning. Thus it becomes important to make a comparative analysis of these algorithms to see which algorithms are suitable for a given planning application. To facilitate this comparison, Kambhampati, et. al. [37], have suggested a generic planning algorithm into which the numerous planning algorithms can be cast.

This generic algorithm is based on planning as *refinement search*: Most planning algorithms work by refining incomplete plans to a final solution. Each incomplete plan stands for the set of solutions it can be refined into. Thus planning becomes searching the space of incomplete plans, pruning those whose set of solutions can be determined to be empty. Planning algorithms mostly differ in their refinement strategies only. Having a generic algorithm allows researchers to compare and contrast these strategies and identify their features.

Kambhampati, et. al., cast several well-known planning algorithms, as well as hybrid combinations of some planning algorithms into their generic algorithm. That way, they could provide preliminary empirical evaluations of various refinement strategies in terms of the structure of their search space, branching factors, overhead involved in performing each type of refinement and the book keeping required.

Although the refinement strategies studied in this work are from state-based planning, the framework itself is general enough to capture the basic properties of HTN planning algorithms at a very abstract level. This work has influenced the high-level search algorithm used by the UMCP system presented in Chapter 6. However, as explained in that chapter, the refinement strategies for HTN planning are rather different.

2.3.3 Complexity Analyses

This section contains the complexity analyses of state-based planning. Complexity analyses determine how much computational resources are required in the worst case for solving a problem.

Decidable, or *recursive* problems are those that can be solved in finite time by some algorithms. *Semidecidable*, or *recursively enumerable* problems are those for which there is no algorithm that always terminates when the input problem has no solution. Semidecidable problems are sometimes, informally referred to as undecidable problems.

2.3.3.1 Chapman's Undecidability Results

In 1987, Chapman [15] did the first analysis of the decidability of state-based planning. His results have been very influential in the planning community; however, there has been a certain amount of confusion about what Chapman's undecidability results actually say, because some of his assumptions are embedded in his proofs. This confusion has been clarified by Erol, et. al [22], as explained in this section.

Chapman's first undecidability theorem ([15, pp. 370–371]) says that all Turing machines with their inputs may be encoded as planning problems in the TWEAK system, and hence planning is undecidable. To prove this theorem, Chapman makes use of the following assumptions:

1. the planning language is function-free;
2. "an infinite [but recursive] set of constants t_i are used to represent the tape squares" [15, p. 371];

3. the initial state is infinite (but recursive). In particular, "there must be countably many **successor** propositions to encode the topology of the tape (and also countably many **contents** propositions to make all but finitely many squares blank)" [15, p. 371].

This theorem is not particularly strong, considering that planning problems seldomly, if ever, involve infinite initial states. In fact, in his discussion of the First Undecidability Theorem [15, p. 344], Chapman says:

> This result is weaker than it may appear ... the proof uses an infinite (though recursive) initial state to model the connectivity of the tape. It may be that if problems are restricted to have finite initial states, planning is decidable. (This is not obviously true though. There are infinitely many constants, and an action can in effect "gensym" one by referring to a variable in its post-conditions that is not mentioned in its preconditions.)

The problem thus introduced was later solved, as will be presented in the following section.

The statement of Chapman's second undecidability theorem is that "planning is undecidable even with a finite initial state if the action representation is extended to represent actions whose effects are a function of their input situations" [15, p. 373].

The meaning of the phrase "effects are a function of their input situations" has caused some confusion. Several researchers thought that Chapman meant operators with context-dependent effects, but an examination the proof of Chapman's theorem makes it clear he is referring to operators that contain function symbols.

2.3.3.2 Complexity Results for STRIPS-style Planning

I have done a complexity analysis of STRIPS-style planning, in collaboration with Dr. Nau and Dr. Subrahmanian [22]. That work serves to clarify some of Chapman's results and provides a rather comprehensive study of how the complexity of STRIPS-style planning varies depending on a number of conditions. The results are summarized in the Tables 2.1, 2.2, and 2.3. Table 2.1 presents the cases where the planning language is allowed to have infinitely many ground terms, either by allowing function symbols, or by allowing infinitely many constant symbols. Table 2.2 presents the cases where the planning language is restricted to contain finitely many ground terms (i.e., no function symbols, finitely many constant symbols). Finally, Table 2.3 presents the cases where the language is further restricted to be propositional (i.e., all predicates are of arity 0, and thus contain no variables). None of the results in these tables are affected when the operators are allowed to contain context-dependent effects.

The following is a brief explanation of the results in the complexity tables.

The decidability results are shown in Table 2.1. If function symbols are allowed, then determining, in general, whether a plan exists is semidecidable. This is true even if we have no delete lists and the precondition list of each operator contains at most one (non-negated) atom. If no function symbols are allowed and only finitely many constant symbols

Table 2.1: Decidability of State-based Planning.

Allow function symbols?	Allow infinitely many constants?	Allow infinite initial states?	Allow delete lists and/or negated preconditions?	Telling whether a plan exists
yes	yes/no	yes/no	yes/no/no$^{\beta}$	semidecidable
	no	no	no$^{\gamma}$	decidable
no	yes	yes	yes/no	semidecidable
		no	yes	semidecidable
			no	decidable
	no	yes/no	yes/no	decidable

$^{\beta}$No operator has more than one precondition.
$^{\gamma}$With acyclicity and boundedness restrictions.

Table 2.2: Complexity of Finite State-based planning.

How are the operators given?	Allow delete lists?	Allow negated preconditions?	Telling whether a plan exists	Telling whether there is a plan of length $\leq k$
given in the input	yes	yes/no	EXPSPACE-comp.	N-EXPTIME-comp.
	no	yes	N-EXPTIME-comp.	N-EXPTIME-comp.
		no	EXPTIME-comp.	N-EXPTIME-comp.
		no$^{\beta}$	PSPACE-complete	PSPACE-complete
fixed in advance	yes	yes/no	PSPACE $^{\delta}$	PSPACE $^{\delta}$
	no	yes	NP $^{\delta}$	NP $^{\delta}$
		no	P	NP $^{\delta}$
		no$^{\beta}$	NLOGSPACE	NP

$^{\beta}$No operator has more than one precondition.
$^{\delta}$With PSPACE- or NP-completeness for some sets of operators.

Table 2.3: Complexity of Propositional State-based planning.

How are the operators given?	Allow delete lists?	Allow negated preconditions?	Telling whether a plan exists	Telling whether there is a plan of length $\leq k$
given in the input	yes	yes/no	PSPACE-complete$^{\epsilon}$	PSPACE-complete
	no	yes	NP-complete$^{\epsilon}$	NP-complete
		no	P $^{\epsilon}$	NP-complete
		no$^{\beta}$/no$^{\gamma}$	NLOGSPACE-comp.	NP-complete
fixed in advance	yes/no	yes/no	constant time	constant time

$^{\beta}$No operator has more than one precondition.
$^{\epsilon}$Results due to Bylander [12].

are allowed, then plan existence is *decidable*, regardless of the presence or absence of delete lists and/or negated preconditions.

Even when function symbols are present, plan existence is decidable if the planning domains being considered have no deletion lists, no negated atoms occur in the precondition list, and the domains satisfy certain acyclicity and boundedness properties.

Whether the planning operators are fixed in advance or given as part of the input, and whether or not they are allowed to have context-dependent effects, does not influence these results.

The complexity results are shown in Tables 2.2 and 2.3. When there are no function symbols and only finitely many constant symbols (so that planning is decidable), the computational complexity varies from constant time to EXPSPACE-complete, depending on the following conditions:

- whether or not we allow delete lists and/or negative preconditions,
- whether or not we restrict the predicates to be propositional (i.e., 0-ary),
- whether we fix the planning operators in advance, or give them as part of the input.

The presence or absence of operators with context-dependent effects does not influence these results.

This work solved an open problem formulated by Chapman in [15]: is planning decidable when the language contains infinitely many constants but the initial state is finite. This problem is decidable in the case where the planning operators have no negative preconditions and no delete lists. If the planning operators are allowed to have negative preconditions and/or delete lists, then the problem is semidecidable.

Chapman's Second Undecidability Theorem states that "planning is undecidable even with a finite initial situation if the action representation is extended to represent actions whose effects are a function of their input situation" [15], i.e., if the language contains function symbols and infinitely many constants. These results show that even with a number of additional restrictions, planning is still undecidable.

This work has also provided equivalence theorems relating definite logic programs to planning with positive, deletion-free operators. This equivalence facilitates transporting many results from logic programming to planning.

Examination of the complexity results reveals several interesting properties:

1. Comparing the complexity of PLAN EXISTENCE in the propositional case (in which all predicates are restricted to be 0-ary) with the *datalog* case (in which the predicates may have constants or variables as arguments, but no function symbols) reveals a regular pattern. In most cases, the complexity in the datalog case is exactly one level harder than the complexity in the corresponding propositional case. We have EXPSPACE-complete versus PSPACE-complete, N-EXPTIME-complete versus NP-complete, and EXPTIME-complete versus polynomial.

2. If delete lists are allowed, then PLAN EXISTENCE is EXPSPACE-complete but PLAN LENGTH is only N-EXPTIME-complete. Normally, one would not expect PLAN LENGTH to be easier than PLAN EXISTENCE. In this case, it happens because the length of a plan can sometimes be doubly exponential in the length of the input. In PLAN LENGTH we are given a bound k, encoded in binary, which confines us to plans of length at most exponential in terms of the input. Hence in the worst case of PLAN LENGTH, finding the plan is easier than in the worst case of PLAN EXISTENCE.

   We do not observe the same anomaly in the propositional case, because the lengths of the plans are at most exponential in the length of the input. Hence, giving an exponential bound on the length of the plan does not reduce the complexity of PLAN LENGTH. As a result, in the propositional case, both PLAN EXISTENCE and PLAN LENGTH are PSPACE-complete.

3. When the operator set is fixed in advance, any operator whose predicates are not all propositions can be mapped into a set of operators whose predicates are all propositions. Thus, planning with a fixed set of datalog operators has basically the same complexity as planning with propositional operators that are given as part of the input.

4. PLAN LENGTH has the same complexity regardless of whether negated preconditions are allowed. This is because what makes the problem hard is how to handle *enabling-condition interactions*, i.e., how to choose operators that achieve several subgoals in order to minimize the overall length of the plan [29], and this task remains equally hard regardless of whether negated preconditions are allowed.

5. Delete lists are more powerful than negated preconditions. Thus, if the operators are allowed to have delete lists, then whether or not they have negated preconditions has no effect on the complexity.

Several other researchers have studied the complexity of STRIPS-style planning. Some of Bylander's results [12] have been displayed in Table 2.3 and indicated as such. Bylander has also studied average-case complexity of propositional STRIPS-style planning under various assumptions as can be found in [14]. Backstrom has done a complexity analysis of a planning representation equivalent to STRIPS representation with similar results [8].

2.3.4 Abstraction Hierarchies

Abstraction hierarchies were originally introduced in the ABSTRIPS [60] system, as described in Section 2.2. ABSTRIPS used heuristic methods in assigning criticality levels to predicates, to determine which of those conditions the planner should try to accomplish first.

Knoblock [41] has developed an algorithm called ALPINE for assigning criticality levels in such a way that during the planning process, establishing a condition (which possibly requires adding new steps to the plan) cannot clobber the previously established conditions with higher criticality levels. This property is called *the ordered monotonicity property.*

ALPINE assigns the criticality levels so that effects of any given action have the same level, and are of same or higher level than the preconditions of the same action. Thus, whenever a new action is inserted to the working plan to achieve a goal condition at that criticality level, the action does not have any effects or preconditions with higher criticality level than the current level. Hence the abstraction hierarchies generated by ALPINE have the ordered monotonicity property. Knoblock's experiments on several planning domains, including "Towers of Hanoi," has indicated that planning using such a hierarchy usually reduces planning time. However, recently, Backstrom [7] has shown that there exist planning problems for which abstraction hierarchies satisfying the ordered monotonicity property may work poorly.

Yang has also worked extensively on abstraction hierarchies, collaborating with a number of researchers. Together with Tenenberg and Woods, Yang has developed ABTWEAK [77], a planning system that utilizes abstraction hierarchies like ABSTRIPS, but it uses TWEAK instead of STRIPS as the underlying plan generator at each level. Yang and Bacchus have identified several properties of abstraction hierarchies as potential indicators of how well they would perform [6]. Particularly,they have identified *the downward refinement property*, which basically states that any solution at an abstraction level can be refined to a final solution, and thus no backtracking across abstraction levels is required. Yang and Bacchus have also studied how the performance of planning systems using abstraction hierarchies are influenced by protecting or not protecting the previously established conditions. Their results indicate that usually, protecting conditions established at the previous level, but not protecting the conditions established at the current level gives better performance.

Abstraction hierarchies, although related, are rather different from task decomposition hierarchies. The similarity and differences among them are discussed in Section 3.6. In fact, as presented in Section 6.5, the UMCP planning system utilizes both of these techniques at the same time.

2.3.5 Partial-order versus Total-order Planning

The first AI planning systems were total order planners; that is, these planners immediately ordered each new step added to a plan with respect to all the other steps. In the following years, most of the planning systems, particularly HTN planning systems, were designed to be partial order: Plan steps were kept partially ordered; new orderings were introduced only to resolve conflicts among steps.

Planning systems that used partial ordering were usually faster than total-order planning systems, but it was not clear whether that was due to using partial order, or to some other feature.

In recent years, several comparative studies of partial order versus total order planning have been done.

One of the first studies on partial versus total order planning was done by Minton et. al. [49]. They designed two planners called TO and UA. TO is a total order planner, which orders each new step in a plan with respect to all the other steps. UA is designed exactly like

TO, except that UA orders a new step only with respect to those steps that has a common effect with the new step. All the steps that may interact are totally ordered in UA; thus it makes more ordering commitments compared to other partial order planning systems. On the other hand, since all interacting steps are ordered, any condition that is possibly true in the plan is also necessarily true, which simplifies planning considerably. Minton, et. al. have shown that UA's search space is never bigger than that of TO, and in some cases it can be exponentially smaller than that of TO.

Barrett and Weld have also studied partial order versus total order planning [10]. Their intuition was that, in some planning problems, a partial order planner like SNLP[46] can easily find the correct order to achieve subgoals while doing conflict resolution, where as a total order planner may need to try many different orderings before finding the right one. Thus, they introduced the concept of *laboriously serializable* goals. Only a few orderings lead to a correct solution for this type of goal. Barrett and Weld designed several artificial planning problems with laboriously serializable goals, and their experiment results corroborate the intuition that partial-order planners perform better than total-order planners in solving problems with this type of goals.

In response to this study, Veloso et. al. [68], has introduced the concept of *linkability*. The degree of linkability in a problem refers to how much backtracking a causal link planner such as SNLP needs to do on the problem, because it has committed to the wrong step to establish a condition. They compare the performance of SNLP to PRODIGY [11], which is a total-order planner, and show that SNLP performs much more slower than PRODIGY in several problems where goals are laboriously linkable.

In recent years, several studies have been made on causal links, and the tradeoffs involved in it [36, 56, 61, 34]. In general, preserving causal links reduces the redundancy in the search space, but increases the branching factor (i.e., the number of ways of establishing a condition). Using causal links may force the planner to commit to the wrong step as an establisher, and cause backtracking.

All these studies on partial versus total ordering has been done on state-based planners, and for good reason. In the HTN framework, total order planning is very restrictive: it prevents interleaving subtasks belonging to different tasks. This issue is discussed in detail in Chapter 4.

2.3.6 Task Networks and Task Decomposition

Very few researchers have tried to integrate task decomposition to their models of planning. Kambhampati and Yang [33, 74] provided the initial steps towards developing a formal model of HTN planning, which the work described in this dissertation has extensively benefited from. Barrett [9] and Young [78] have incorporated task decomposition techniques to their planning systems as a tool for increasing the speed of state-based planning. The general idea was to encapsulate heuristic fast ways of accomplishing conditions via task decompositions. In that aspect, their approach is similar to that of MACROPS [23] developed by Fikes et. al. which used predefined sequences of operators (i.e. macros) to provide shortcuts in the

search space.

2.4 Discussion

As presented throughout Section 2.3, extensive research effort has gone into formalizing and analyzing the ideas embedded in planning systems. These studies provided a very good understanding of state-based planning, but they did not cover many important aspects of more application-oriented planning systems, particularly task decomposition and critics. As I will explain in Chapter 3, the concepts of task networks and task decomposition bring a totally new dimension to planning, with a more powerful and natural representation, and a broader set of algorithmic concerns. This dissertation complements the previous studies by formalizing and analyzing the HTN planning techniques, and by introducing new, provably correct HTN planning algorithms.

The formal framework of HTN planning, presented in Chapter 3, is based on the ideas embedded in NOAH, NONLIN and DEVISER planning systems, although many modifications and extensions have been necessary to make the formal framework consistent, coherent, and comprehensive. This part of the dissertation is analogous to the work on the STRIPS language to free it from semantic flaws [45].

The second part of the dissertation provides a complexity analysis of HTN planning to give insight into the algorithmic difficulties and bottlenecks, analogous to the complexity analyses on state-based planning, described in Section 2.3.3. This part also contains an expressivity analysis of the HTN language in comparison with the STRIPS language, which proves that the HTN representation is strictly more powerful. This study is unique perhaps because there have not been many planning languages formalized to sufficient detail to be compared. In fact, part of this work involves developing the necessary mathematical tools and definitions for comparing the expressivity of planning languages.

The remaining chapters of the dissertation have focused on developing provably correct, efficient planning algorithms for HTN planning, which is in some ways analogous to what Chapman [15] and McAllester [46] accomplished for state-based planning. First, an abstract, but provably correct planning algorithm is developed. Then detailed algorithms are devised for computing the steps of the abstract algorithm. This part of the work also involves designing correct critics mechanisms for HTN planning, based on constraint satisfaction. Causal links introduced in SNLP [46] are modeled as a special type of constraint; they are employed for establishing conditions and preserving systematicity.

Chapter 3

HTN Formalism

This chapter presents the HTN formalism I have developed. The first section contains an informal description of HTN planning, intended to give an intuitive feel for the formalism presented in the subsequent sections. Section 3.2 describes the syntactic constructs of HTN planning. The next two sections present the model-theoretic and operational semantics for these syntactic constructs. Section 3.5 presents a provably correct HTN planning algorithm. Finally, Section 3.6 describes how various HTN concepts in previous HTN planning systems are modelled in this framework.

3.1 An Overview of HTN planning

One of the motivations for HTN planning was to close the gap between STRIPS-style planning, and the operations-research techniques for project management and scheduling [64]. As a result, there are some similarities between HTN planning and STRIPS-style planning, but also some significant differences.

HTN planning representations for actions and states of the world are similar to those used in STRIPS-style planning[1], which was described in Section 2.3.1. Thus, each state of the world is represented by the set of atoms true in that state. Actions, which are called *primitive tasks* in the HTN terminology, are represented using STRIPS-style operators.

The primary difference between HTN planners and STRIPS-style planners is in what they plan for, and how they plan for it. In STRIPS-style planning, the objective is to find a sequence of actions that will bring the world to a state that satisfies certain conditions or "attainment goals." Planning usually proceeds by finding operators that have the desired effects, and by making the preconditions of those operators into subgoals. In contrast, HTN planners search for plans that accomplish *task networks*, which can include things other than

[1]The terms "STRIPS-style planning" and "state-based planning" are interchangeably used to refer to any planner (either total- or partial-order) in which the planning operators are STRIPS-style operators (i.e., operators consisting of three lists of atoms: a precondition list, an add list, and a delete list). These atoms are normally assumed to contain no function symbols.

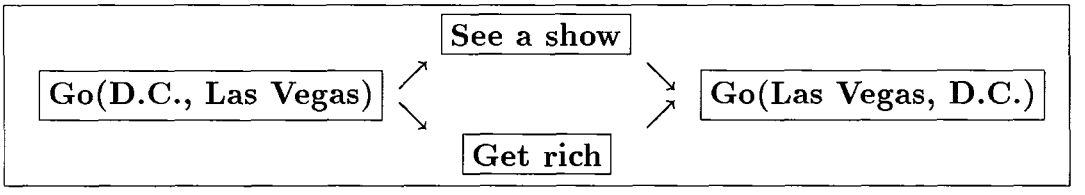


Figure 3.1: A task network

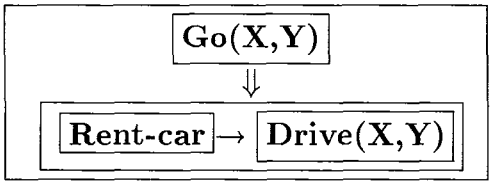


Figure 3.2: A (simplified) method for going from X to Y.

just attainment goals; and they plan via task decomposition and conflict resolution, which shall be explained shortly.

A task network is a collection of tasks that need to be carried out, together with constraints on the order in which tasks can be performed, the way variables are instantiated, and what literals must be true before or after each task is performed. For example, Figure 3.1 contains a task network for a trip to Las Vegas. Unlike STRIPS-style planning, the constraints may or may not contain conditions on what must be true in the final state.

A task network that contains only primitive tasks is called a *primitive task network.* Such a network might occur, for example, in a scheduling problem. In this case, an HTN planner would try to find a schedule (task ordering and variable bindings) that satisfies all the constraints.

In the more general case, a task network can contain *non-primitive tasks*, which the planner needs to figure out how to accomplish. Non-primitive tasks cannot be executed directly, because they represent activities that may involve performing several other tasks. For example the task of traveling to New York can be accomplished in several ways, such as flying, driving or taking the train. Flying would involve tasks such as making reservations, going to the airport, buying ticket, boarding the plane; and flying would only work if certain conditions were satisfied, such as availability of tickets, being at the airport on time, having enough money for the ticket, and so forth.

Ways of accomplishing non-primitive tasks are represented using constructs called *methods.* A method is a syntactic construct of the form (α, d) where α is a non-primitive task, and d is a task network. It states that one way to accomplish the task α is to achieve all the tasks in the task network d without violating the constraints in d. Figure 3.2 presents a (simplified) method for accomplishing **Go(X,Y)**.

A number of different systems that use heuristic algorithms have been devised for HTN planning [64, 69, 73], and several recent papers have tried to provide general descriptions of

1. Input a planning problem **P**.
2. If **P** contains only primitive tasks, then
   resolve the conflicts in **P** and return the result.
   If the conflicts cannot be resolved, return failure.
3. Choose a non-primitive task t in **P**.
4. Choose an expansion for t.
5. Replace t with the expansion.
6. Use critics to find the interactions among the tasks in **P**,
   and suggest ways to handle them.
7. Apply one of the ways suggested in step 6.
8. Go to step 2.

Figure 3.3: The basic HTN Planning Procedure.

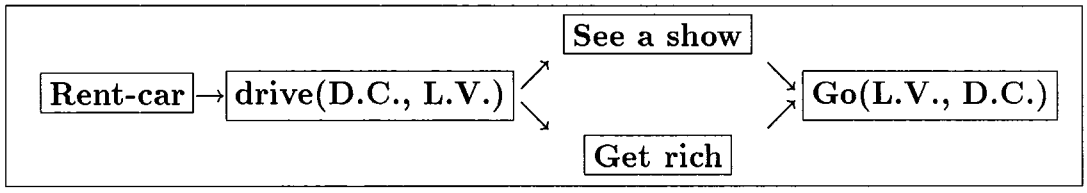


Figure 3.4: A decomposition of the the task network in Fig. 3.1

these algorithms [74, 33]. Figure 3.3 presents the essence of these algorithms. As shown in this figure, HTN planning works by expanding tasks and resolving conflicts iteratively, until a conflict-free plan can be found that consists only of primitive tasks.

Expanding each non-primitive task (steps 3–5) is done by finding a method capable of accomplishing the non-primitive task, and replacing the non-primitive task with the task network produced by the method. For example, the task **Go(D.C., Las Vegas)** in the task network of Figure 3.1 can be expanded using the method in Figure 3.2, producing the task network in Figure 3.4.

The task network produced in Step 5 may contain conflicts caused by the interactions among tasks. For example, in Figure 3.4, if we use up all our money in order to rent the car, we may not be able to see a show. The job of finding and resolving such interactions is performed by critics. As explained in Section 2.2.7, critics were introduced into NOAH [58] to identify and deal with several kinds of interactions (not just deleted preconditions) among the different networks used to reduce each non-primitive operator. This is reflected in Steps 6 and 7 of Figure 3.3: after each reduction, a set of critics is checked so as to recognize and resolve interactions between this and any other reductions. Thus, critics provide a general mechanism for detecting interactions early, so as to reduce the amount of backtracking.

Section 6.3 contains a detailed description of how to detect and resolve interactions in HTN planning in such a way to preserve soundness, completeness, and systematicity.

In contrast to STRIPS-style attainment goals, task networks are much richer in structure. Any sequence of actions for which the attainment goals are true in the final state constitute a plan in the STRIPS paradigm. There is very little control over which actions can be used in the plan, what must be true in the intermediate states, or in which order the goals must be accomplished. This control can be exercised only by modeling the application domain at a very fine granularity, duplicating many operators, and inventing many new preconditions for them, on a per problem basis. On the other hand, HTN planning provides full control over the actions in a plan: only those actions that appear in the methods for the task (and methods for its subtasks) can be used. Restrictions on the task orderings or intermediate states can easily be expressed using the HTN constraints. This expressivity of the HTN constructs can be very useful in many application domains. For example, consider an automated manufacturing application. A process plan for producing a part usually involves many machining operations with several constraints on the way these operations must be performed. Such a process plan can be represented as a method. It may be possible to represent this domain in the STRIPS paradigm; however, for a planner to be able to deduce the constraints from the domain description, the domain must be modelled at an unnecessarily detailed level (possibly to the level of physical interactions, and chemical processes), which makes it computationally prohibitive and unfeasible.

3.2 Syntax for HTN Planning

A language $\mathcal{L}$ for HTN planning is a first-order language with some extensions.

Definition 10 The vocabulary of $\mathcal{L}$ is a tuple $\langle V, C, P, F, T, N \rangle$, where V is an infinite set of variable symbols, C is a finite set of constant symbols, P is a finite set of predicate symbols, F is a finite set of *primitive*-task symbols (denoting actions), T is a finite set of *compound*-task symbols, and N is an infinite set of symbols used for labeling tasks. All these sets of symbols are mutually disjoint.

Note that the HTN language does not contain any function symbols. Function symbols were introduced to planning in situation calculus to model actions. HTN planning uses primitive tasks for this purpose. First order predicate logic includes function symbols, and contains mechanisms for reasoning about them. However, planning systems seldom aspire to reason about functions. More often it is the case that the interpretation of any function is fixed and known in advance. Such a function can be modelled as a relation using a predicate symbol, or by using procedural attachment (i.e., by providing a piece of code that actually computes the function).

Note also that the set of constant symbols is restricted to be finite. Most planning applications involve only a finite number of objects that can be identified and manipulated. When the application involves infinite sets of objects such as numbers, these can be dealt

with more efficiently using procedural mechanisms instead of reasoning about them using axiomatic approaches.

Most planning algorithms work with finitely many constants. TWEAK [15] is a particular exception, which relies on the domain to contain infinitely many constant symbols in order to operate correctly. As was discussed in Section 2.2, TWEAK assumes the set of constants to be infinite so that a consistent ground instantiation of a partial plan can be computed in polynomial time. If the set of possible values for each variable is finite, then finding a consistent ground instantiation is NP-complete. However, the price for this reduction in a single step is very dear: as presented in Section 2.3.3, allowing infinitely many constants makes planning undecidable!

Definition 11 A *state* is a list of ground atoms.

The atoms appearing in that list are said to be true in that state and those that do not appear are false in that state. Thus the notion of states in HTN planning is the same as in state-based planning, which was presented in Section 2.3.1.

Definition 12 A *primitive task* has the form $do[f(x_1, \ldots, x_k)]$, where $f \in F$ and $x_1, \ldots, x_k$ are terms.

Primitive tasks in HTN planning correspond to actions in STRIPS-style planning. Effects and preconditions of primitive tasks are declared using operators.

Definition 13 An *operator* has the form
[operator $f(v_1, \ldots, v_k)$
:pre $l_1, \ldots, l_m$
:post $l'_1, \ldots, l'_n$
].

f is a primitive task symbol, and $l_1, \ldots, l_m$ are literals describing when f is executable. $l'_1, \ldots, l'_n$ are literals describing the effects of f, and $v_1, \ldots, v_k$ are the variable symbols appearing in the literals.

Note that this syntax is slightly different from that of STRIPS operators presented in Section 2.3.1. For syntactic convenience, atoms in the delete list are negated and merged with the atoms in the add list into a single postcondition list.

Definition 14 A *plan* is a sequence of ground primitive tasks.

Up to this point, the syntax and the semantics of the constructs defined are very similar to those in STRIPS-style planning, which was described in Section 2.3.1. The difference of HTN planning is in what we plan for. STRIPS-style planners plan for "attainment goals", which are conditions that must be made true in the world. Any plan that makes those conditions true in its final state is considered a valid solution, no matter which actions are used or what the intermediate states contain. HTN planners search for plans that accomplish more complex behaviors, which are denoted as tasks and task networks.

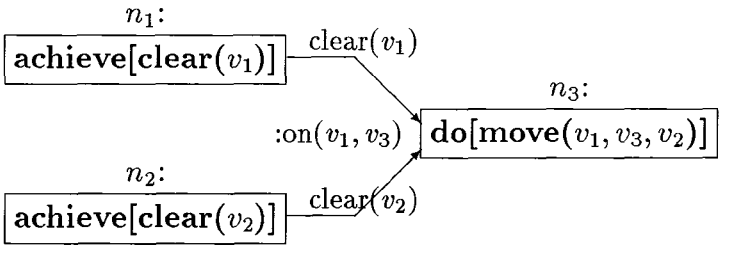


Figure 3.5: Graphical representation of a task network.

Definition 15 A *goal task* has the form $achieve[l]$, where l is a literal. A *compound task* has the form $perform[t(x_1, \ldots, x_k)]$, where $t \in T$ and $x_1, \ldots, x_k$ are terms.

Both goal tasks and compound tasks are referred to as *non-primitive tasks.*

Definition 16 A *task network* has the form $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$, where each α_i is a task; each $n_i \in N$ is a label for α_i (to distinguish it from any other occurrences of α_i in the network); and ϕ is a boolean formula constructed from atomic HTN constraints. ϕ can contain the usual logic connectives conjunction, disjunction, and negation.

HTN planning has a rather rich set of atomic constraint types:

- Variable binding constraints are of the form $(v_1 = c)$, $(v_1 = v_2)$. $v_1, v_2 \in V$ are variable symbols, and $c \in C$ is a constant symbol.

- Ordering constraints are usually of the form $(n \prec n')$ where $n, n' \in N$. As to be described precisely in the semantics part, such an expression denotes that the task labeled with n must finish before the task labeled with n' starts, thus all the subtasks in the decomposition of n must precede all the subtasks in the decomposition of n'.

  In the more general case, instead of individual node labels from N, ordering constraints can contain *node expressions* of the form $first[n_i, n_j, \ldots]$ or $last[n_i, n_j, \ldots]$ to refer to the task that starts first and to the task that ends last among a set of tasks, respectively.

- State constraints are of the form (n, l), (l, n), and (n, l, n'), where $v, v' \in V$, l is a literal, $c \in C$. n and n' can be either node labels in N, or they can be node expressions as was described in the previous paragraph. Intuitively (this will be formalized in Chapter 3.2, (n, l), (l, n) and (n, l, n') mean that l must be true in the state immediately after n, immediately before n, and in all states between n and n', respectively.

- A special type of state constraint is $(initially\ l)$. This constraint denotes that the literal l must be true in the initial state.

As an example, Fig. 3.6 gives a formal representation of the task network shown in Figure 3.5. In this blocks-world task network there are three tasks: clearing v_1, clearing v_2, and moving v_1 to v_2. The task network also includes the constraints that moving v_1 must

$$[(n_1 : achieve[clear(v_1)])(n_2 : achieve[clear(v_2)])(n_3 : do[move(v_1, v_3, v_2)])$$
$$(n_1 \prec n_3) \wedge (n_2 \prec n_3) \wedge (n_1, clear(v_1), n_3) \wedge (n_2, clear(v_2), n_3) \wedge$$
$$(on(v_1, v_3), n_3) \wedge \neg(v_1 = v_2) \wedge \neg(v_1 = v_3) \wedge \neg(v_2 = v_3)]$$

Figure 3.6: Formal representation of the task network of Fig. 3.5.

be done last, that v_1 and v_2 must remain clear until v_1 is moved, that v_1, v_2, v_3 are different blocks, and that $on(v_1, v_3)$ be true immediately before v_1 is moved. Note that $on(v_1, v_3)$ appears as a constraint, not as a goal task. The purpose of the constraint $(on(v_1, v_3), n_3)$ is to ensure that v_3 is bound to the block under v_1 immediately before the move. Representing $on(v_1, v_3)$ as a goal task would mean moving v_1 onto some block v_3 before it is moved onto v_2, which is not what is intended.

Definition 17 A task network containing only primitive tasks is called a *primitive task network.*

Definition 18 A *method* is a construct of the form (α, d) where α is a non-primitive task, and d is a task network.

As defined formally in Section 3.2, this construct means that one way of accomplishing the task α is to accomplish the task network d, i.e. to accomplish all the subtasks in the task network without violating the constraint formula of the task network. For example, a blocks-world method for achieving $on(v_1, v_2)$ would look like $(achieve(on(v_1, v_2)), d)$, where d is the task network in Fig. 3.6.

Every method for a goal task has the implicit constraint that the goal condition must become true in the end. Every goal task also has an implicit method of the form $(achieve[l], [(n : do[f]), (l, n)])$ which contains only one dummy primitive task f with no effects, and the constraint that the goal l is true immediately before $do[f]$. Thus, if the goal condition is already true, the goal task has an empty expansion.

Each primitive task has exactly one operator for it, where as a non-primitive task can have an arbitrary number of methods.

Definition 19 A *planning domain* $\mathcal{D}$ is a pair $\langle Op, Me \rangle$, where Op is a list of operators, and Me is a list of methods.

Definition 20 A *planning problem instance* $\mathbf{P}$ is a triple $\langle d, I, \mathcal{D} \rangle$, where $\mathcal{D}$ is a planning domain, I is the initial state, and d is the input task network to plan for.

$solves(\sigma, d, I)$ is a syntactic construct which is used to mean that σ is a plan for the task network d at state I.

The language of $\mathbf{P}$ is the HTN language $\mathcal{L}$ generated by the constant, predicate, and task symbols appearing in $\mathbf{P}$, along with an infinite set of variables and an infinite set of node

labels. Thus, the set of constants, predicates and tasks are all part of the input to an HTN planner.

Next, let us define some restrictions on HTN-planning problems. **P** is *primitive* if the input task network d contains only primitive tasks. This corresponds to the case where the planner is used only for scheduling. **P** is *regular* if all the task networks in the methods and d contain at most one non-primitive task, and that non-primitive task is ordered with respect to all the other tasks in the network. Surprisingly, this class of HTN-planning problems is closely related to STRIPS-style planning, as we shall see in Chapter 5. **P** is *propositional* if no variables are allowed. **P** is *totally ordered* if all the tasks in any task network are totally ordered.

Definition 21 PLAN EXISTENCE is the following problem: given $\mathbf{P} = \langle d, I, \mathcal{D} \rangle$, is there a plan that solves **P**?

3.3 Model-Theoretic Semantics

3.3.1 Semantic Structure

Definition 22 A semantic structure for HTN planning is a triple

$$M = \langle \mathcal{S}_M, \mathcal{F}_M, \mathcal{T}_M \rangle$$

The subscript M is omitted whenever the model is clear from context. $\mathcal{S}$, $\mathcal{F}$, and $\mathcal{T}$ are described below.

$\mathcal{S} = 2^{\{\text{all ground atoms}\}}$ is the set of states. Each state in $\mathcal{S}$ is a set, consisting of the atoms true in that state. Any atom not appearing in a state is considered to be false in that state. A state corresponds to a "snapshot" instance of the world.

$\mathcal{F} : F \mathbf{x} C^* \mathbf{x} \mathcal{S} \rightarrow \mathcal{S}$ is a partial function for interpreting the actions. Given a primitive task symbol from F, with ground parameters from C, and an input state, $\mathcal{F}$ tells us which state we would end up with, if we were to execute the action. For a given action, $\mathcal{F}$ might be undefined for some input states, namely those for which the action is not executable.

Up to this point, the semantics for HTN planning coincides with the semantics for STRIPS-style planning. Both involve states which are represented as sets of atoms, and actions (i.e. primitive tasks) which map one state to another state.

HTN planning takes STRIPS-style planning one step further by introducing non-primitive tasks, and task networks.

$\mathcal{T} : \{\text{ground non-primitive tasks}\} \rightarrow 2^{\{\text{ground primitive task networks}\}}$ is a function that maps each non-primitive task α to a (not necessarily finite) set of ground primitive task networks $\mathcal{T}(\alpha)$. Each primitive task network d in $\mathcal{T}(\alpha)$ describes one way of accomplishing α.

There are two restrictions on the way $\mathcal{T}()$ interprets a goal task $achieve[l]$. First, l must be true at the end of any task network in $\mathcal{T}(achieve[l])$. Second, since an empty plan can be

used to accomplish a goal task if the goal literal is already true, $\mathcal{T}(achieve[l])$ must contain a task network consisting of a single dummy task with the constraint that l is true.

Modeling each non-primitive task as a set of primitive task networks may seem unnecessarily complicated at first. The straightforward alternative would be to model each task as the set of plans that achieve that task. While this alternative would work well for the attainment goals of STRIPS-style planning, such a model does not contain enough information for the purpose of HTN planning. Ways of accomplishing non-primitive tasks cannot be identified by the conditions true in the final state. In HTN planning, each way of accomplishing a task must contain two types of information:

1. The set of actions that must be executed to accomplish the task,
2. The constraints on the way these actions must be executed. This information is necessary so that, when the planning problem involves accomplishing multiple tasks (most planning problems do), the actions needed to accomplish each task can be correctly combined together without any harmful interactions.

Both type of information is nicely encapsulated in a primitive task network.

A non-primitive task networks is modeled as a set of primitive task networks, similar to the way non-primitive tasks are modeled. A model does not need to interpret a task network directly, the interpretation of the task network can be constructed from the interpretations of its constituent tasks. For this purpose, $\mathcal{T}$ is extended to $\overline{\mathcal{T}}$ as follows:

- $\overline{\mathcal{T}}(\alpha) = \{[(n : \alpha), TRUE]\}$, if α is a ground primitive task. Thus, a ground primitive task is mapped to a primitive task network containing only that primitive task.
- $\overline{\mathcal{T}}(\alpha) = \mathcal{T}(\alpha)$, if α is a ground non-primitive task.
- $\overline{\mathcal{T}}(\alpha) = \bigcup_{\alpha' \textit{ is a ground instance of } \alpha} \overline{\mathcal{T}}(\alpha')$, if α is a task containing variables.
- $\overline{\mathcal{T}}(d) = \{d\}$, if d is a ground primitive task network.
- Let $d = [(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$ be a ground task network possibly containing non-primitive tasks. Then

$$\overline{\mathcal{T}}(d) = \{compose(d_1, \ldots, d_m, \phi) \mid d_i \in \overline{\mathcal{T}}(\alpha_i),\ i = 1 \ldots m\}.$$

  This definition follows the intuition that the set of solutions to a task network consists of combinations of the solutions to the tasks in that task network.

  compose is defined as follows. Suppose, for $i == 1..m$

$$d_i = [(n_{i1} : \alpha_{i1}) \ldots (n_{ik_i} : \alpha_{ik_i}), \phi_i]$$

.

Then[2]

$$compose(d_1, \ldots, d_m, \phi) = [(n_{11} : \alpha_{11}) \ldots (n_{mk_m} : \alpha_{mk_m}), \phi_1 \wedge \ldots \phi_m \wedge \phi'],$$

where ϕ' is obtained from ϕ by making the following replacements:

- replace $(n_i < n_j)$ with $(last[n_{i1}, \ldots, n_{ik_i}] < first[n_{j1}, \ldots, n_{jk_j}])$, since all tasks in the decomposition of n_i must precede all tasks in the decomposition of n_j;
- replace (l, n_i) with $(l, first[n_{i1}, \ldots, n_{ik_i}])$, since l needs to be true immediately before the first task in the decomposition of n_i;
- replace (n_i, l) with $(last[n_{i1}, \ldots, n_{ik_i}], l)$;
- replace (n_i, l, n_j) with $(last[n_{i1}, \ldots, n_{ik_i}], l, first[n_{j1}, \ldots, n_{jk_j}])$;
- everywhere that n_i appears in ϕ in a $first[]$ or a $last[]$ expression, replace it with $n_{i1}, \ldots, n_{ik_i}$.

These replacements in the constraint formula of the original task network d are required because the node labels in the constraint formula are not meaningful once the associated tasks are expanded. Thus any references to those node labels must be properly replaced with the node labels in the expansion.

- $\overline{\mathcal{T}}(d) = \bigcup_{d' \text{ is a ground instance of } d} \overline{\mathcal{T}}(d')$, if d is a task network containing variables.

In subsequent sections and chapters, the bar in $\overline{\mathcal{T}}$ is omitted, and only $\mathcal{T}$ is used instead.

3.3.2 Satisfaction

This section describes how syntactic expressions such as operators and methods take truth values in a given model. We will use the phrases "... is true in model M" and "... is satisfied by model M" interchangeably.

A model M satisfies an operator o, if M interprets the primitive task associated with the operator so that the primitive task is executable under the conditions specified in the preconditions of the operator, and has the effects specified in the postconditions of the operator. Thus:

Definition 23 An operator [operator $f(v_1, \ldots, v_k)$:pre $l_1 \ldots l_m$:post $l'_1 \ldots l'_n$] is *satisfied* by a model M iff for any ground substitution θ and any state s, $\mathcal{F}_M$ has the following properties, where E_n, E_p are the sets of negative and positive literals in $l'_1, \ldots, l'_n$, respectively:

- if $l_1\theta, \ldots, l_m\theta$ are true in s, then $\mathcal{F}_M(f, v_1\theta, \ldots, v_k\theta, s) = (s - E_n\theta) \cup E_p\theta$;
- otherwise, $\mathcal{F}_M(f, v_1\theta, \ldots, v_k\theta, s)$ is undefined.

[2]The variables and node labels in each task network d_i must be renamed (standardized) so that no common variable or node label occurs.

Note that the definition of satisfaction for operators in the HTN paradigm and in the STRIPS paradigm are very similar.

In order to relate tasks in a primitive task networks to plans, a mapping is required, as defined below:

Definition 24 Let M be a model, d = $[(n_1 : \alpha_1) \cdots (n_{m'} : \alpha_{m'}), \phi]$ be a ground primitive task network, s_0 be a state, and

$$\sigma = (f_1(c_{11}, \ldots, c_{1k_1}), \ldots, f_m(c_{m1}, \ldots, c_{mk_m}))$$

be a plan executable at s_0. Thus, $s_i = \mathcal{F}_M(f_i, c_{i1}, \ldots, c_{ik_i}, s_{i-1})$ for $i = 1 \ldots m$, which are the intermediate states, are all well-defined. A *matching* π from d to σ is defined to be a one-to-one function from $\{1, \ldots, m'\}$ to $\{1, \ldots, m\}$ such that whenever $\pi(i) = j$, $\alpha_i = do[f_j(c_{j1}, \ldots, c_{jk_j})]$.

Thus a matching provides a total ordering on the tasks.

Now, let us define the conditions under which a model M satisfies $solves(\sigma, d, s)$, that is, the conditions under which σ is a plan that accomplishes the task network d starting at state s, in M. Let us first consider the case where d is primitive.

Definition 25 M satisfies $solves(\sigma, d, s)$ if $m = m'$, and there exists a one-to-one onto matching π that makes the constraint formula ϕ true. The constraint formula is evaluated as follows:

- $(c_i = c_j)$ is true, if c_i, c_j are the same constant symbols;
- $first[n_i, n_j, \ldots]$ evaluates to $min\{\pi(i), \pi(j), \ldots\}$;
- $last[n_i, n_j, \ldots]$ evaluates to $max\{\pi(i), \pi(j), \ldots\}$;
- $(n_i \prec n_j)$ is true if $\pi(i) < \pi(j)$;
- (l, n_i) is true if l holds in $s_{\pi(i)-1}$;
- (n_i, l) is true if l holds in $s_{\pi(i)}$;
- (n_i, l, n_j) is true if l holds for all s_e, $\pi(i) \leq e < \pi(j)$;
- logical connectives $\neg, \wedge, \vee$ are evaluated as in propositional logic.

Definition 26 Let d be a task network, possibly containing non-primitive tasks. A model M satisfies $solves(\sigma, d, s)$ if for some task network $d' \in \mathcal{T}_M(d)$, M satisfies $solves(\sigma, d', s)$.

For a method (α, d) to be satisfied by a given model, not only must any plan for d also be a plan for α, but in addition, any plan for a task network tn containing d must be a plan for the task network obtained from tn by replacing d with α. To precisely explain this notion, we introduce the following definition:

Definition 27 Given a model M, two sets of ground primitive task networks TN and TN', TN is said to *cover* TN', iff for any state s, any plan σ executable at s, and any $d' \in TN'$, the following property holds:

> Whenever there exists a matching π between d' and σ such that σ at s satisfies the constraint formula of d', then there exists a $d \in TN$ such that for some matching π' with the same range as π, σ at s makes the constraint formula of d true.

Intuitively, a set of task networks TN covers another set of task networks TN', if any time part of a plan achieves a task network in TN', the same portion of the plan also achieves a task network in TN.

Definition 28 A method (α, d) is satisfied by a model M, iff $\mathcal{T}_M(\alpha)$ *covers* $\mathcal{T}_M(d)$.

Thus, accomplishing d always accomplishes α, even in the presence of other tasks and constraints. Definitions for covering and the satisfiability of methods are a bit complicated, because the meaning of a task network not only involves a characterization of the plans that solves it in isolation, but also in the presence of other tasks and constraints.

Definition 29 A model M satisfies a planning domain $\mathcal{D} = \langle Op, Me \rangle$, if M satisfies all operators in Op, and all methods in Me.

3.4 Operational Semantics

A plan σ solves a planning problem $\mathbf{P} = \langle d, I, \mathcal{D} \rangle$ if any model that satisfies $\mathcal{D}$ also satisfies $solves(\sigma, d, I)$. However, given a planning problem, how do we find plans that solve it?

Definition 30 Let d be a primitive task network containing only primitive tasks, and let I be the initial state. A plan σ is a *completion* of d at I, denoted by $\sigma \in comp(d, I, \mathcal{D})$, if σ is executable (i.e. the preconditions of each action in σ are satisfied) and σ corresponds to a total ordering of the primitive tasks in a ground instance of d that satisfies the constraint formula of d. Satisfaction of the constraint formula was defined in the previous section. For non-primitive task networks d, $comp(d, I, \mathcal{D})$ is defined to be $\emptyset$.

Definition 31 Let d be a non-primitive task network that contains a (non-primitive) node $(n : \alpha)$. Let $m = (\alpha', d')$ be a method, and θ be the most general unifier of α and α'. We define $reduce(d, n, m)$ to be the task network obtained from $d\theta$ by replacing $(n : \alpha)\theta$ with the task nodes of $d'\theta$, modifying the constraint formula ϕ of $d'\theta$ into ϕ' (as we did for *compose*), and combining with the constraint formula of $d'\theta$ with a conjunct.

We denote the set of reductions of d by $red(d, I, \mathcal{D})$. Reductions correspond to *task decomposition*. A plan σ solves a primitive task network d at initial state I iff $\sigma \in comp(d, I, \mathcal{D})$; a plan σ solves a non-primitive task network d at initial state I iff σ solves some reduction $d' \in red(d, I, \mathcal{D})$ at initial state I.

Definition 32 The set of plans $sol(d, I, \mathcal{D})$ that solves a planning problem instance $\mathbf{P} =< d, I, \mathcal{D} >$ is defined iteratively as:

$$\begin{array}{lcl} sol_1(d, I, \mathcal{D}) & = & comp(d, I, \mathcal{D}) \\ sol_{n+1}(d, I, \mathcal{D}) & = & sol_n(d, I, \mathcal{D}) \cup \bigcup_{d' \in red(d,I,\mathcal{D})} sol_n(d', I, \mathcal{D}) \\ sol(d, I, \mathcal{D}) & = & \cup_{n<\omega} sol_n(d, I, \mathcal{D}) \end{array}$$

Intuitively, $sol_n(d, I, \mathcal{D})$ is the set of plans that can be derived in n steps, and $sol(d, I, \mathcal{D})$ is the set of plans that can be derived in any finite number of steps.

Section 3.3 has presented a model-theoretic semantics for HTN planning, and this section has presented an operational, fixed-point semantics that provides a procedural characterization of the set of solutions to planning problems. The next step is to show that the model-theoretic semantics and operational semantics are equivalent, so that we can use the model-theoretic semantics to get a precise understanding of HTN planning, and use the operational semantics to build sound and complete planning systems. The following theorem states that $sol(d, I, \mathcal{D})$ is indeed the set of plans that solves $\langle d, I, \mathcal{D} \rangle$.

Theorem 1 (Equivalence Theorem) Given a task network d, an initial state I, and a plan σ, σ is in $sol(d, I, \mathcal{D})$ if and only if any model that satisfies $\mathcal{D}$ also satisfies $solves(\sigma, d, I)$.

Proof. [$\rightarrow$]. Since $sol(d, I, \mathcal{D})$ is defined recursively in terms of completions and reductions, it suffices to show that

(a)Given any primitive task-network d, if $\sigma \in comp(d, I, \mathcal{D})$, then any model that satisfy $\mathcal{D}$ also satisfies $solves(\sigma, d, I)$,

(b) Given any model M which satisfies $\mathcal{D}$, any two task networks d, d' such that $d' \in red(d, I, \mathcal{D})$, whenever M satisfies $solves(\sigma, d', I)$, then it also satisfies $solves(\sigma, d, I)$.

(a). Assume $\sigma \in comp(d, I, \mathcal{D})$. Then, σ is a totally ordered ground instance of d that makes the constraint formula true. Let M be any model that satisfies $\mathcal{D}$. Because M satisfies the operators, $\mathcal{F}_M$ will project the intermediate states to be exactly the same as projected in the completion. Furthermore, the constraint formula is evaluated in the same way both for finding completions and for determining whether a model satisfies $solves(\sigma, d, I)$. Thus, given a primitive task network d, and a model M that satisfies $\mathcal{D}$, M satisfies $solves(\sigma, d, I)$ iff $\sigma \in comp(d, I, \mathcal{D})$.

(b). Let d' be in $red(d, I, \mathcal{D})$. Then there exists a method m, and a task node n in d such that $d' = reduce(d, n, m)$. Let M be a model that satisfies $\mathcal{D}$, and also $solves(\sigma, d', I)$.

Without loss of generality, let's assume d, m, d' have the following forms:

$$\begin{array}{lll} d = & & [(n_1 : \alpha_1) \ldots (n_k : \alpha_k)\ \ (n : \alpha), \phi], \\ m = & (\alpha, & [(n'_1 : \alpha'_1) \ldots (n'_j : \alpha'_j), \psi]), \\ d' = & & [(n'_1 : \alpha'_1) \ldots (n'_j : \alpha'_j)\ \ (n_1 : \alpha_1) \ldots (n_k : \alpha_k), \phi' \wedge \psi]. \end{array}$$

Since M satisfies $solves(\sigma, d', I)$, there exists
$d'_1 \in \mathcal{T}(\alpha'_1), \ldots, d'_j \in \mathcal{T}(\alpha'_j), d_1 \in \mathcal{T}(\alpha_1), \ldots, d_k \in \mathcal{T}(\alpha_k)$ such that σ is a plan for $compose(d_1, \ldots, d'_j, d_1, \ldots, d_k, \phi' \wedge \psi)$.

Thus, there exists a matching π between σ
and $compose(d'_1, \ldots, d'_j, d_1, \ldots, d_k, \phi' \wedge \psi)$, such that the constraint formula of $compose(d'_1, \ldots, d'_j, \psi)$, which correspond to the portion of the task network corresponding to the expansion of α, is satisfied. From this fact and that M satisfies the method m, we conclude there exists a $d'' \in \mathcal{T}(\alpha)$ and a matching π' such that σ makes the constraint formula of d'' true.

Consider $compose(d_1, \ldots, d_k, d'', \phi) \in \mathcal{T}(d)$. Construct a matching π'' by extending π' to $d_1, \ldots, d_k$ (taking the same value as π for those places). σ satisfies ϕ and the constraints of $d_1, \ldots, d_k, d''$. Thus M satisfies $solves(\sigma, d, I)$.

[$\leftarrow$]. We will show that whenever $\sigma \notin sol(d, I, \mathcal{D})$, there exists a model M that satisfies $\mathcal{D}$, but not $solves(\sigma, d, I)$.

Here is how we construct $M =< \mathcal{S}, \mathcal{F}, \mathcal{T} >$:

- $\mathcal{S} = 2^{\{ground\ atoms\}}$.
- $\mathcal{F}(f, c_1 \ldots, c_k, s) = (s - N\theta) \cup P\theta$, whenever the operator for f is of the form $(f(v_1, \ldots, v_k), l_1, \ldots, l_k)$, where θ is the substitution $\{c_i/v_i | 1 \leq i \leq k\}$, and N, P are the sets of negative and positive literals in $\{l_1, \ldots, l_k\}$, respectively.
- $\mathcal{T}(\alpha) = \{d | d$ is a ground instance of a primitive task-network obtained from α by a finite number of reductions$\}$, for any non-primitive task α.

When $\mathcal{T}$ is extended to cover task networks as defined in Section 3.2, we observe that $\mathcal{T}(d) = \{d' | d'$is a ground instance of a primitive task-network obtained from d by a finite number of reductions$\}$ for any task network d.

$\mathcal{F}$ is defined such that M satisfies all the operators in $\mathcal{D}$. By definition of $\mathcal{T}$, whenever $d' \in red(d, I, \mathcal{D})$, $\mathcal{T}(d') \subseteq \mathcal{T}(d)$. Thus $\mathcal{T}(d)$ covers $\mathcal{T}(d')$ and M satisfies all the methods in $\mathcal{D}$.

Given a primitive task network d, and a model M that satisfies $\mathcal{D}$, M satisfies $solves(\sigma, d, I)$ iff $\sigma \in comp(d, I, \mathcal{D})$

Let d be a primitive task network such that $\sigma \notin sol(d, I, \mathcal{D})$. In that case, $\sigma \notin comp(d, I, \mathcal{D})$, either. In part (a) of the proof, we showed that for any model that satisfies the operators, $comp(d, I, \mathcal{D})$ is exactly the set of plans that solves the primitive task network d. Thus we conclude M does not satisfy $solves(\sigma, d, I)$.

Consider the alternative where d is a non-primitive task network such that $\sigma \notin sol(d, I, \mathcal{D})$. Assume M satisfies $solves(\sigma, d, I)$. Then there exists a primitive task network $d' \in \mathcal{T}(d)$ such that σ is a plan for d'. From part (a) of the proof, σ must be in $comp(d', I, \mathcal{D})$. However, since $\mathcal{T}(d)$ contains only primitive task networks that can be obtained by a finite number of reductions from d, we conclude $\sigma \in sol(d, I, \mathcal{D})$, which is a contradiction. ■

procedure *UMCP*:
1. Input a planning problem $\mathbf{P} = \langle d, I, \mathcal{D} \rangle$.
2. if d is primitive, then
   If $comp(d, I, \mathcal{D}) \neq \emptyset$, return a member of it.
   Otherwise return FAILURE.
3. Pick a non-primitive task node $(n : \alpha)$ in d.
4. Nondeterministically choose a method m for α.
5. Set $d := reduce(d, n, m)$.
6. Set $\Gamma := \tau(d, I, \mathcal{D})$.
7. Nondeterministically set $d :=$ some element of Γ.
8. Go to step 2.

Figure 3.7: UMCP: Universal Method-Composition Planner

3.5 UMCP: A Provably Correct HTN Algorithm

Using the syntax and semantics developed in the previous section, the HTN planning procedure presented in Figure 3.3 can be formalized. Figure 3.7 presents this planning algorithm, called UMCP (for Universal Method-Composition Planner).

It should be clear that *UMCP* mimics the definition of $sol(d, I, \mathcal{D})$, except for Steps 6 and 7 (which correspond to the critics). As discussed before, HTN planners typically use their critics for detecting and resolving interactions among tasks (expressed as constraints) in task networks at higher levels, before all subtasks have been reduced to primitive tasks. By eliminating some task orderings and variable bindings that lead to dead ends, critics help prune the search space. In UMCP, this job is performed by the critic function τ. τ takes as input an initial state I, a task network d, and a planning domain $\mathcal{D}$; and produces as its output a set of task networks Γ. Each member of Γ is a candidate for resolving some[3] of the conflicts in d. Several restrictions are required on τ to make sure it preserves soundness, completeness and systematicity:

1. If $d' \in \tau(d, I, \mathcal{D})$ then $sol(d', I, \mathcal{D}) \subseteq sol(d, I, \mathcal{D})$.

   Thus, any plan for d' must be a plan for d ensuring soundness.

2. If $\sigma \in sol_k(d, I, \mathcal{D})$ for some k, then there exists $d' \in \tau(d, I, \mathcal{D})$ such that $\sigma \in sol_k(d', I, \mathcal{D})$.

   Thus, whenever there is a plan for d, there is a plan for some member d' of $\tau(d, I, \mathcal{D})$. In addition, if the solution for d is no further than k expansions, so is the solution for d'. The latter condition ensures that τ does not create infinite loops by undoing previous expansions.

[3]It might be impossible or too costly to resolve some conflicts at a given level, and thus handling those conflicts can be postponed.

3. For any two different task networks $d_1, d_2 \in \tau(d, I, \mathcal{D})$, it must be the case that $sol(d_1, I, \mathcal{D}) \cap sol(d_2, I, \mathcal{D}) = \emptyset$.

   Thus the set of solutions for each task network in $\tau(d, I, \mathcal{D})$ are disjoint, in order to preserve systematicity. This condition does not need to be strictly enforced, as it does not affect the correctness of the planning algorithm. However, enforcing it may make the planning algorithm more efficient.

In contrast to the abundance of well understood STRIPS-style planning algorithms (such as [24, 15, 46, 35]), HTN planning algorithms have typically not been proven to be sound or complete. However, using the formalism presented in Sections 3.2 and 3.3, the soundness and completeness of UMCP can be established.

The soundness and completeness results follow directly from the equivalence theorem using the fact that $UMCP$ directly mimics $sol()$. The restrictions on the critic function ensure that τ does not introduce invalid solutions and that it does not eliminate valid solutions.

Theorem 2 (Soundness) *Whenever $UMCP$ returns a plan, it achieves the input task network at the initial state with respect to all the models that satisfy the methods and the operators.*

Proof. Assume that on input $\mathbf{P} =< d, I, \mathcal{D} >$, UMCP halts in n iterations, and returns σ. Using induction on n, we prove that $\sigma \in sol(d, I, \mathcal{D})$, and from the equivalence theorem we conclude that any model that satisfies $\mathcal{D}$ also satisfies $solves(\sigma, d, I)$.

[Base case: $n = 0$.] d must be a primitive task network and $\sigma \in comp(d, I, \mathcal{D})$. Thus, $\sigma \in sol(d, I, \mathcal{D})$.

[Induction Hypothesis] Assume for $n < k$ if UMCP returns σ in n iterations, then $\sigma \in sol(d, I, \mathcal{D})$.

Suppose on input $\mathbf{P} =< d, I, \mathcal{D} >$ UMCP halts in $n = k$ iterations. and returns σ. Then d is a non-primitive task network. Let $d_1 = reduce(d, n, m)$ be the value assigned to d at step 5 of UMCPin the first iteration, and let $d_2 \in \tau(d_1, I, \mathcal{D})$ be the value assigned to d at step 7 of UMCPin the first iteration.

On input $\mathbf{P} =< d_2, I, \mathcal{D} >$, the planner halts in $k - 1$ steps and returns σ. Thus, by induction hypothesis, $\sigma \in sol(d_2, I, \mathcal{D})$. From restriction 1 on the critic function $\tau()$ and from $d_2 \in \tau(d_1, I, \mathcal{D})$ we conclude $\sigma \in sol(d_1, I, \mathcal{D})$. Since $d_1 \in red(d, I, calD)$, from the definition of $sol()$ we conclude $\sigma \in sol(d, I, \mathcal{D})$. ■

Theorem 3 (Completeness) *Whenever $UMCP$ fails to find a plan, there is no plan that achieves the input task network at the initial state with respect to all the models that satisfy the methods and the operators.*

Proof. Assume $\sigma \in sol(d, I, \mathcal{D})$. Let k be the minimum number of reductions needed to derive σ; i.e. $\sigma \in sol_{k+1}(d, I, \mathcal{D})$, but $\sigma \notin sol_k(d, I, \mathcal{D})$. We show that there exists a sequence of non-deterministic choices such that the UMCP halts in k iterations and returns a plan.

Proof by induction on k.

[Base case: $k = 0$.] In that case $\sigma \in comp(d, I, \mathcal{D})$, and UMCP finds a plan in in step 2.

[Induction Hypothesis] Assume whenever the number of reductions needed to derive σ is less then j (i.e. $k < j$), there exists a sequence of non-deterministic choices for which UMCP returns a plan in k iterations.

Let $\sigma \in sol((d, I, \mathcal{D})$, and suppose it takes exactly j reductions to derive σ. Let $(n : \alpha)$ be the node picked by UMCP in step 3 in the first iteration (Since this is not a non-deterministic choice, the planner could pick any non-primitive node at that step.) Let m be the method used for reducing $(n : \alpha)$ in the derivation. Let $d_1 = reduce(d, n, m)$. $reduce()$ is defined in Section 3.2 such that the order of reductions does not matter; i.e. For a task network d that contains two non-primitive tasks n_1, n_2 with corresponding methods m_1, m_2,

$$reduce(reduce(d, n_1, m_1), n_2, m_2) \quad and \quad reduce(reduce(d, n_2, m_2), n_1, m_1)$$

are equal modulo variable and node label names. Thus, since we obtained d_1 from d by reducing n with the same method used in the derivation, we conclude $\sigma \in sol_{j-1}((d_1, I, \mathcal{D})$.

As $\sigma \in sol_{j-1}((d_1, I, \mathcal{D})$, by the second restriction on $\tau()$, there exists $d_2 \in \tau(d_1, I, \mathcal{D})$ such that $\sigma \in sol_{j-1}((d_2, I, \mathcal{D})$. Let the planner non-deterministically choose d_2 to assign to d at step 7 in the first iteration. By induction hypothesis, the planner will halt in $j - 1$ more iterations and return a plan. ■

3.6 Discussion

The syntax and semantics defined in this chapter can be used to explain a number of issues that are often discussed in the literature.

3.6.1 Tasks and Task-decomposition

There appears to be some general confusion about the nature and role of tasks in HTN planning. This seems largely due to the fact that HTN planning emerged, without a formal description, in implemented planning systems [58, 64]. Many ideas introduced in HTN planning (such as nonlinearity, partial order planning, etc.) were formalized only as they were adapted to STRIPS-style planning, and only within that context [15, 46, 49, 16, 35]. The following subsections discuss the view of tasks in this dissertation, and possible alternative views.

3.6.1.1 A View of Tasks as Jobs

The formalism described in this chapter views tasks as activities we need to plan, jobs that need to be accomplished. Each method tells us one way of achieving a task, and it also

tells us under which conditions that way is going to succeed (as expressed in the constraint formula). The task decomposition refers to choosing a method for the task, and using it to achieve the task. For example, possible methods for the task of traveling to a city might be flying (under the condition that airports are not closed due to bad weather), taking the train (under the condition that there is an available ticket), or renting a car (under the condition that I have a driver's license). Tasks and task networks provide a more natural and effective way of representing planning problems than STRIPS-style attainment goals. Lansky [44] has promoted this opinion for action-based planning in general.

3.6.1.2 A View of Tasks as Efficiency Hacks

One view of HTN planning totally discards compound tasks, and views methods for goal tasks as heuristic information on how to go about achieving the goals (i.e., which operator to use, in which order to achieve the preconditions of that operator). Although this is a perfectly coherent view, it is rather restrictive and there is more to HTN planning, as demonstrated in Chapter 5, where the expressivity results are presented.

3.6.1.3 A View of Tasks as Action Abstraction

Yang and Kambhampati [74, 33] have viewed tasks as high-level actions. High-level actions have preconditions and effects, just as regular STRIPS operators; however, they can be expanded into lower level actions. Planning proceeds iteratively: goals in the planning problem are established using the highest-level actions first. Once the harmful interactions among those actions are resolved by adding proper ordering and variable binding restrictions and causal links, the actions in the working plan are expanded. The iteration is repeated until all the actions are primitive and all the conflicts are resolved.

Each iteration of the procedure described above creates a layer of abstraction, in some ways similar to the abstraction layers produced by ABSTRIPS [60]. Unlike the abstraction hierarchies of ABSTRIPS, which are based on preconditions, this type of abstraction is based on actions. ABSTRIPS can provide efficiency gains by reducing the number of preconditions that need to be considered at each level; this type of action hierarchies can provide efficiency gains by reducing the number of actions that need to be considered at each level. Furthermore, the expansions may provide heuristic information as to how the preconditions or subactions must be ordered.

This approach is appealing in many aspects; however, the distinction between high-level actions and primitive actions are rather obscure and may cause confusion. Primitive actions are atomic, and they always have the same effect on the world; high-level actions can be decomposed into a number of primitive actions, and the effect of executing a high-level action depends not only on the methods chosen for doing decompositions, but also on the interleavings with other actions. For example, consider the task of "round-trip to New York". The amount of money I have got after the trip depends on whether I flew or took a train, and also on my activities in New York (night clubs, etc).

Good ABSTRIPS-style abstraction hierarchies can be automatically generated for state-based planning problems, as was discussed in Section 2.3.4. However, because of the difficulties described above, it is rather difficult to generate good action hierarchies for a given state-based planning problem. In particular, abstraction hierarchies only influence the speed of a planner, but action hierarchies can affect the correctness of a planner, in addition to its efficiency.

Action hierarchy approach to planning may be considered as a more principled version of the view of tasks and task decomposition as efficiency hacks. Both views are considerably different from the view of HTN planning presented in this dissertation. Unlike high-level actions, tasks are not "executed" for their effects in the world, but they are ends in themselves. In this aspect, tasks are more similar to STRIPS-style goals than STRIPS-style actions.

3.6.1.4 Task Decomposition and Abstraction Hierarchies

The basic idea in abstraction hierarchies is to tell a planning system which part of the problem to work on at each iteration, more particularly, which preconditions to establish and which preconditions to ignore at a given iteration. The same idea can also be applied to HTN planning.

Recall our previous discussion that tasks are similar to goals in certain aspects. An HTN planner, at each iteration, needs to decide which task to expand next. This corresponds to Step 3 in the UMCP algorithm displayed in Figure 3.7. The choice of which task to expand next can be made arbitrarily, but an intelligent choice will improve the performance of a planning system significantly.

There is a clear need for algorithms that will process the domain specifications and assign criticality levels to tasks in such a way to improve performance. There is already a considerable body of work on generating good abstraction hierarchies in state-based planning, as discussed in Section 2.3.4, which can form a starting point.

Chapter 5 shows that state-based planning problems are a special case of HTN problems. That chapter also contains a polynomial transformation from state-based planning problems to HTN problems which maps each predicate to a goal task. I conjecture that if a given abstraction hierarchy works well in a state-based planning problem, the same criticality assignment will work well on a translation of this problem to HTN framework. Since the HTN framework is much more complex than state-based planning, the work on abstraction hierarchies must be significantly extended to work well on all HTN planning problems.

3.6.1.5 Distinctions among Tasks and Attainment Goals

Here are some more examples to further clarify the distinctions among different types of tasks and STRIPS-style goals. Building a house requires many other tasks to be performed (laying the foundation, building the walls, etc.), thus it is a compound task. It is different from the goal task of "having a house," since buying a house would achieve this goal task, but not the compound task of building a house (the agent must build it himself). As another example, the compound task of making a round trip to New York cannot easily be expressed as a

single goal task, because the initial and final states would be the same. Goal tasks are very similar to STRIPS-style goals. However, in STRIPS-style planning, *any* sequence of actions that make the goal expression true is a valid plan, where as in HTN planning, only those plans that can be derived via decompositions are considered as valid. This allows the user to rule out certain undesirable sequences of actions that nonetheless make the goal expression true. For example, consider the goal task of "being in New York", and suppose the planner is investigating the possibility of driving to accomplish this goal, and suppose that the agent does not have a driver's license. Even though learning how to drive and getting a driver's license might remedy the situation, the user can consider this solution unacceptable, and while writing down the methods for **be-in(New York)**, add the constraint that the method of driving succeeds only when the agent already has a driver's license.

3.6.2 High-Level Effects

Typically, HTN planners allow one to attach "high-level" effects to subtasks in methods, similar to the way we attach effects to primitive tasks using operators. Some HTN-planners such as NONLIN assume that the high-level effects will be true immediately after the corresponding subtasks, even if they are not asserted by any primitive tasks. This is problematic: one can obtain the same sequence of primitive tasks with different tasks and methods, and given high-level effects, the final state might depend on what particular task(s) the sequence was intended for.

Yang [74] addresses this problem by attaching high-level effects to tasks directly using operators. In addition, he requires that for each high-level effect l associated with a task α, every decomposition of α must contain a subtask with the effect l, which is not clobbered by any other subtask in the same decomposition. However, this solution does not preclude the possibility that l might be clobbered by actions in the decompositions of other tasks.

In the framework presented in this dissertation, only primitive tasks can change the state of the world; non-primitive tasks are not allowed to have direct effects. Instead, high-level effects are represented as constraints of the form (n, l) so that the planner verifies those effects to hold. Thus the previous problem can be avoided, but a planning system can still benefit from guiding search with high-level effects (one of the primary reasons they are often used in implemented planning systems).

Another concern is what happens to a high-level effect when the task it is associated with gets expanded. The natural choice is to attach the high-level effect to the last task in the expansion. However, the last task may be impossible to identify, if the task ordering in the expansion is not sufficiently constrained. A similar problem arises when a task has a condition that needs to be true before the task, and the task gets expanded. In that type of situations, most planning systems take a rather arbitrary ad-hoc approach.

The framework introduced in this dissertation specifically contains the forms $first[n_1, \ldots, n_k]$ and $last[n_1, \ldots, n_k]$ so that one can refer to the the first task that starts and the last task that ends among a set of tasks, respectively. Section 3.3 gives a precise description of how to modify the constraint formula at task expansion to correctly handle

all types of constraints, including those that represent high-level effects.

3.6.3 Conditions

HTN planners often allow several types of conditions in methods. For instance NONLIN has *use-when, supervised,* and *unsupervised* conditions.

Supervised conditions, similar to preconditions in STRIPS-style planning, are those that the planner needs to accomplish, thus in our framework, they appear as goal nodes in task networks. In the task network shown in Fig. 3.5, the conditions **clear(v_1)** and **clear(v_2)** appear as goal tasks for that reason.

Unsupervised conditions are conditions that are needed for achieving a task but supposed to be accomplished by some other part of the task network (or the initial state). For example, a **load(package,vehicle)** task in a transport logistics domain would require the package and vehicle to be at the same location, but it might be the responsibility of another task (e.g. a vehicle dispatcher) to accomplish that condition. Thus, the load task must only verify the condition to be true and it must not try to achieve the condition by task decompositions, or insertions of new actions. In the HTN framework described in this dissertation, unsupervised conditions are represented as state constraints so that an HTN planner will seek variable bindings/task orderings that would make those conditions true, but it will not try to establish those conditions by inserting new actions or doing task decompositions. NONLIN ignores the unsupervised conditions until all tasks are expanded into primitive tasks, which is not always an efficient strategy. On the other hand, the UMCP planning system presented in Chapter 6 tries to process unsupervised conditions at higher levels (if possible to do so without compromising correctness of the planner) to prune the search space.

HTN planners employ filter conditions (called use-when conditions in NONLIN) for deciding which methods to try for a task expansion and reduce the branching factor by eliminating irrelevant methods. For example consider the task of going to New York, and the method of accomplishing it by driving. One condition necessary for this method to succeed is having a driver's license. Although a driver's license can be obtained by learning how to drive and going through the paperwork, the user of the planner might consider this unacceptable, and in that case he would specify having a driver's license not as a goal task but as a filter condition , and the method of driving to New York would not be considered if the agent does not have a driver's license at the appropriate point in the plan.

In [16], Collins and Pryor state that filter conditions are ineffective. They argue that filter conditions do not help pruning the search space for partial order planners, because it is not possible to check whether they hold or not in an incomplete plan. They also empirically demonstrate that ignoring the filter conditions until all the subgoals are achieved is quite inefficient. Although their study of filter conditions is in the context of STRIPS representation, to a large extend it also applies to HTN planning. NONLIN, for instance, evaluates filter conditions as soon as they are encountered, and unless it can establish those conditions to be necessarily true, it will backtrack. Thus, NONLIN sometimes backtracks over filter conditions which would have been achieved by actions in later task expansions.

Hence NONLIN is not complete. Although it might not always be possible to determine whether a filter condition is true in an incomplete plan, filter conditions can still be used to prune the search space. Our framework represents filter conditions as state constraints and planning algorithms based on this framework can employ constraint satisfaction/propagation techniques to prune inconsistent task networks. For example if a task network contains the filter condition (l,n), and also another constraint $(n_1, \neg l, n_2)$, one can deduce that n should be either before n_1, or after n_2. Furthermore, some filter conditions might not be affected by the actions (e.g. conditions on the type of objects), and thus it suffices to check the initial state to evaluate those. This kind of filter condition can also be very helpful in pruning the search space.

Tate [65] also promotes the use of filter conditions, but suggests this must be done very carefully to work efficiently, without compromising correctness.

In summary, the HTN constraints provide a principled way of representing many kinds of conditions, and UMCP employs techniques for using them effectively without sacrificing soundness or completeness.

3.6.4 Critics and Constraints

One attractive feature of HTN systems is that by using various sorts of critics, they can handle many different kinds of interactions, thus allowing the analysis of potential problems in plans, and preventing later failure and backtracking. However, this mechanism has been little explored in the formal literature, primarily due to the procedural nature of handling interactions between subtasks.

The formalism described in this chapter has identified the conditions a critic function must satisfy in order to preserve soundness, completeness and systematicity. These conditions has provided the guidelines in devising correct critics mechanisms presented in Section 6.3.

Designing a correct critic mechanism is rather challenging. HTN planning provides a large number of constraint types, and each type of constraint must be handled with care. The challenge is compounded by the fact that critics most of the time deal with non-primitive task networks; they would not be of much use otherwise. In evaluating a constraint, all possible refinements of a task network must be considered, and preferably this must be done without explicitly enumerating all of those possible refinements. Chapter 6 presents the details of how UMCP planning system meets these challenges.

The constraint formula in a method for a task mostly serves to identify and resolve interactions from other tasks; however,another important use of constraints is encoding control information. When the user encodes a domain, it might be known in advance that certain variable bindings, task orderings etc. lead to dead ends. These can be eliminated by posting constraints (in methods) so that the planner does not waste time deriving this information. This ability to encode known shortcuts and/or pitfalls was offered as a major motivation for the move to procedural networks in NOAH [58].

Chapter 4

Complexity of HTN Planning

The HTN formalism presented in Chapter 3 makes it possible to analyze the complexity of HTN planning under a number of conditions:

- whether the tasks in task networks are required to be totally ordered,
- whether variables are allowed,
- whether domain specification is fixed in advance, or part of the input.
- Whether non-primitive tasks are allowed, and whether they must be totally ordered with respect to other tasks.

Table 4.1 contains a summary of this analysis. The theorem statements and discussions on their implications appear in the following sections. The proofs of the theorems are presented in the appendix.

4.1 Undecidability Results

It is easy to see that HTN planning problems can simulate any context-free grammar by using primitive tasks to emulate terminal symbols, compound tasks to emulate non-terminal symbols, and methods to encode grammar rules. More interesting is the fact that HTN planning problems can simulate any two context-free grammars, and constraints in the methods can enforce the existence of a solution to the planning problem if and only if these two grammars have a common string in their corresponding languages. Whether the intersection of the languages of two context-free grammars is non-empty is a semi-decidable problem [32]. Thus:

Theorem 4 PLAN EXISTENCE is strictly semi-decidable, even if **P** is restricted to be propositional, and all the methods are restricted to be totally ordered and to contain at most two tasks.

Table 4.1: Complexity of HTN Planning

Restrictions on non-primitive tasks	Must every HTN be totally ordered?	Are variables allowed?	
		no	yes
none	no	Undecidable$^{\alpha}$	Undecidable$^{\alpha\beta}$
	yes	in EXPTIME; PSPACE-hard	in DEXPTIME; EXPSPACE-hard
"regularity" (≤ 1 non-primitive task, which must follow all primitive tasks)	doesn't matter	PSPACE-complete	EXPSPACE-complete$^{\gamma}$
no non-primitive tasks	no	NP-complete	NP-complete
	yes	Polynomial time	NP-complete

$^{\alpha}$Decidable with acyclicity restrictions.
$^{\beta}$Undecidable even when the planning domain is fixed in advance.
$^{\gamma}$In PSPACE when the planning domain is fixed in advance, and PSPACE-complete for some fixed planning domains.

This result might seem surprising at first, since the state space (i.e., the number and size of states) is finite. If the planning problem were that of finding a path from the initial state to a goal state (as in STRIPS-style planning), indeed it would be decidable, because, for that problem, whenever there is a plan, there is also a plan that does not go through any state twice, and thus only a finite number of plans need to be examined. On the other hand, HTN planning can represent compound tasks accomplishing which might require going through the same state many times, which explains this result.

Instead of encoding each context-free grammar rule as a separate method, it is possible to encode these rules with predicates in the initial state, and to have a method containing variables and constraints such that only those decompositions corresponding to the grammar rules encoded in the initial state are allowed. Hence, even when the domain description (i.e., the set of operators and methods) is fixed in advance, it is possible to find planning domains for which planning is undecidable, as stated in the following theorem:

Theorem 5 There are HTN planning domains that has only totally ordered methods each with at most two tasks, for which PLAN EXISTENCE is strictly semi-decidable.

4.2 Decidability and Complexity Results

One way to make PLAN EXISTENCE decidable is to restrict the methods to be acyclic. In that case, each task can be expanded up to only a finite depth, and thus the problem becomes decidable. More formally, a *k-level-mapping* is defined to be a function *level*() from ground instances of tasks to the set $\{0, \ldots, k\}$, such that for any method that can expand a ground

task α to a task network containing a ground task α', $level(\alpha) > level(\alpha')$. Furthermore, $level(\alpha)$ must be 0 for every primitive task α.

Intuitively, $level()$ assigns levels to each ground task, and makes sure that tasks can be expanded into only lower level tasks, establishing an acyclic hierarchy. In this case, any task can be expanded to a depth of at most k. Therefore,

Theorem 6 PLAN EXISTENCE is decidable if **P** has a k-level-mapping for some integer k.

Examples of such planning domains can be found in manufacturing, where the product is constructed by first constructing the components and then combining them together.

Another way to make PLAN EXISTENCE decidable is to restrict the interactions among the tasks. Restricting the task networks to be totally ordered limits the interactions that can occur between tasks. Tasks need to be achieved serially, one after the other; interleaving subtasks for different tasks is not possible. Thus interactions between the tasks are limited to the input and output state of the tasks, and the "protection intervals", i.e the literals that need to be preserved, which are represented by state constraints of the form (n, l, n'). This constraint states that the literal l needs to remain true from the end of the task labeled with n till the start of the task labeled with n'.

Under the above restrictions, it is possible to create a table with an entry for each task, input/output state pair, and set of protected literals, that tells whether each task can be accomplished starting at the corresponding input state and ending at the output state without ever making the protected literals false. Using dynamic programming techniques, the entries in the table can be computed in DOUBLE-EXPTIME, or in EXPTIME if the problem is further restricted to be propositional. Chapter 5 presents several polynomial transformations from STRIPS-style planning problems to totally ordered HTN planning problems. These transformations can be combined with the complexity results on STRIPS-style planning in [22, 12] to establish a lower bound on the complexity of totally-ordered HTN planning. Thus:

Theorem 7 PLAN EXISTENCE is EXPSPACE-hard and in DOUBLE-EXPTIME if **P** is restricted to be totally ordered. PLAN EXISTENCE is PSPACE-hard and in EXPTIME if **P** is further restricted to be propositional.

Recall that in regular HTN planning problems, any task network –both the initial input task network, and those we obtain by expansions– can contain at most one non-primitive task. Thus, subtasks in the expansions of different tasks cannot be interleaved, which is similar to what happens in Theorem 7. But in Theorem 7, there could be several non-primitive tasks in a task network, and we needed to keep track of all of them (which is why a table was used). If the planning problem is regular, we only need to keep track of a single non-primitive task, its input/final states, and the protected literals. Since the size of a state is at most exponential, the problem can be solved in exponential space. But even with regularity and several other restrictions, it is still possible to reduce the EXPSPACE-complete STRIPS-style planning problem (described in [22]) to the HTN planning. Thus:

Theorem 8 PLAN EXISTENCE is EXPSPACE-complete if **P** is restricted to be regular. It is still EXPSPACE-complete if **P** is further restricted to be totally ordered, with at most one non-primitive task symbol in the planning language, and all task networks containing at most two tasks.

When the problem is further restricted to be propositional, the complexity goes down one level:

Theorem 9 PLAN EXISTENCE is PSPACE-complete if **P** is restricted to be regular and propositional. It is still PSPACE-complete if **P** is further restricted to be totally ordered, with at most one non-primitive task symbol in the planning language, and all task networks containing at most two tasks.

When the planning domain $\mathcal{D}$ is fixed in advance, the number of ground atoms and ground tasks is polynomial in the length of the input to the planner, and thus the complexity of planning with a fixed planning domain is no harder than the complexity of propositional planning. It is proven in [22] that there exists STRIPS-style planning domains for which planning is PSPACE-complete. Those domains can be transformed into regular HTN planning domains for which planning is PSPACE-complete. Hence:

Theorem 10 If **P** is restricted to be regular and $\mathcal{D}$ is fixed in advance, then PLAN EXISTENCE is in PSPACE. Furthermore, there exists fixed regular planning domains $\mathcal{D}$ for which PLAN EXISTENCE is PSPACE-complete.

Suppose a planning problem is primitive, and either propositional or totally ordered. Then the problem's membership in NP is easy to see: nondeterministically guess a total ordering and variable binding (obviously of polynomial size), and then check whether the constraint formula on the task network is satisfied. Furthermore, unless the planning problem is restricted to be both totally-ordered and propositional, the constraint language can be used to represent the satisfiability problem, and thus the problem is NP-hard. Hence:

Theorem 11 PLAN EXISTENCE is NP-complete if **P** is restricted to be primitive, or primitive and totally ordered, or primitive and propositional. However, PLAN EXISTENCE can be solved in polynomial time if **P** is restricted to be primitive, totally ordered, and propositional.

4.3 Discussion

The complexity results on HTN planning have been summarized in Table 4.1. These results show how the complexity of solving HTN planning problems depend on various factors. A number of conclusions can be drawn from these results:

1. HTN planning is undecidable under even a very severe set of constraints. In particular, it is undecidable even if no variables are allowed, as long as there is the possibility that a task network can contain two non-primitive tasks without specifying the order in which they must be performed.

2. In general, what restrictions are put on the non-primitive tasks has a bigger effect on complexity than whether or not we allow variables, or require tasks to be totally ordered.

3. Placing restrictions either on non-primitive tasks or on the ordering of tasks makes planning decidable. If either restriction is removed individually, planning remains decidable, but removing both simultaneously makes planning undecidable.

4. If there are no restrictions on non-primitive tasks, then whether or not we require tasks to be totally ordered has a bigger effect (namely, decidability vs. undecidability) than whether or not we allow variables. But in the presence of restrictions on non-primitive tasks, whether or not we allow variables has a bigger effect than whether or not we require tasks to be totally ordered.

These results show that handling interactions among non-primitive tasks is the most difficult part of HTN planning. In particular, if subtasks in the expansions for different tasks can be interleaved, then planning is undecidable, even if no variables are allowed. Limiting the interactions among tasks significantly reduces the complexity of planning. For example, preventing subtask interleavings by restricting task networks to be totally ordered makes HTN planning to EXPTIME. Limiting the interactions among subtasks by allowing at most one non-primitive task in a task network further reduces the complexity to PSPACE.

Complexity results must be interpreted cautiously. They are worst-case results. For instance, a problem is PSPACE-complete means that, any algorithm that solves this problem correctly will consume memory polynomial in the size of the input for *some* problem instances. Any given algorithm may consume much less memory for most problem instances. These worst case results show the computational requirements for solving the hardest problem that can be represented in the HTN language. Most problems encountered by a planning system may not be among those hardest problems. Nonetheless, complexity results provide valuable insight into the structure of a problem and the associated algorithmic difficulties. The complexity analysis of HTN planning has guided the algorithm development described in Chapter 6.

Chapter 5

Expressivity of HTN Planning

Prior to the formalization of HTN planning described in Chapter 3, it was not clear what kind of planning problems can be represented and solved via HTN planning. In particular, it has long been a topic of debate whether HTN planning is merely an "efficiency hack" over STRIPS-style planning, or HTN planning is actually more expressive. This chapter addresses these question formally.

There is not a well established definition of expressivity for planning languages. It is possible to define expressivity based on model-theoretic semantics, operational semantics, and even on the computational complexity of problems that can be represented in the planning language. Section 5.1 presents definitions of expressivity for each of those three cases, and Section 5.2 contains the theorems that HTN planning is strictly more expressive than STRIPS-style planning according to all three of them.

5.1 Expressivity Definitions for Planning

5.1.1 Complexity-based Expressivity

One criterion for evaluating the expressivity of a language is the computational complexity of solving problems represented in that language. One might ask: "How difficult are the most difficult problems that can be represented in this language?" If the set of problems that can be represented in a language $\mathcal{L}_1$ belongs to a higher complexity class than the set of problems that can be represented in another language $\mathcal{L}_2$, then $\mathcal{L}_1$ might be considered more expressive than $\mathcal{L}_2$. As computational complexity is measured in terms of the size of the input problem instance, it also means that $\mathcal{L}_1$ is more concise than $\mathcal{L}_2$.

The theory of computational complexity [26] is based on transformations (also called reductions) between sets of problem instances. A transformation τ is a mapping from a set of problem instances P_1 to a set of problem instances P_2 such that any problem $p_1 \in P_1$ has a solution iff $\tau(p_1)$ also has a solution. Transformations are required to be computable in polynomial time when the two classes of problems at hand are decidable; otherwise transformations are simply required to be computable (i.e. computable in finite time on a Turing

machine).

The following is a definition for complexity-based expressivity, which uses the notion of transformations:

Definition 33 (Complexity-based Expressivity) A planning language $\mathcal{L}_1$ can be expressed by a planning language $\mathcal{L}_2$, iff there exists a transformation from the set of problem instances represented in $\mathcal{L}_1$ to those that can be represented in $\mathcal{L}_2$.

When a planning language $\mathcal{L}_1$ can be expressed by a planning language $\mathcal{L}_2$, but not vice versa, then $\mathcal{L}_2$ is strictly more expressive than $\mathcal{L}_1$.

Transformations can also be considered as translations between planning languages. Whenever there is a transformation from $\mathcal{L}_1$ to $\mathcal{L}_2$, a planner for $\mathcal{L}_2$ can solve problems represented in $\mathcal{L}_1$ by first translating them into $\mathcal{L}_2$.

5.1.2 Model-Theoretic Expressivity

Baader [5] has presented a model-theoretic definition of expressivity for knowledge representation languages. The following is a description of Baader's definition of expressivity and how it can be adapted for planning languages.

Baader's notion of expressivity is based on the idea that if a language L_1 can be expressed by another language L_2, then for any set of sentences Γ_1 in L_1, there must be a corresponding set of sentences Γ_2 in L_2 with the same meaning. Baader captures "the same meaning" by requiring the two set of sentences to have the same set of semantic models. Because L_2 might contain symbols not necessary for expressing L_1, and the name of the symbols used must not make a difference, it suffices for the set of models for Γ_1 and the set of models for Γ_2 be equivalent modulo a symbol translation function ω.

More formally, given a function ω from the set of symbols in L_1 to the set of symbols in L_2, two models M_1 and M_2 are defined to be equivalent module ω, denoted as $M_1 =^{\omega} M_2$ iff for any symbol s in L_1, $M_1(s) = M_2(\omega(s))$. In other words, for any symbol $s \in L_1$, s and $\omega(s)$ must have the same interpretation.

The equivalence between models can be extended to sets of models as follows: Two sets of models $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent modulo a symbol translation function ω, denoted by $\mathcal{M}_1 =^{\omega} \mathcal{M}_2$, iff for any model $M_1 \in \mathcal{M}_1$, there exists a model $M_2 \in \mathcal{M}_2$ such that $M_1 =^{\omega} M_2$, and for any model $M_2 \in \mathcal{M}_2$, there exists a model $M_1 \in \mathcal{M}_1$ such that $M_1 =^{\omega} M_2$.

Definition 34 A knowledge representation language L_1 can be expressed by another knowledge representation language L_2, iff there exists a function ψ from the set of sentences in L_1 to the set of sentences in L_2, and a symbol translation function ω from the set of symbols in L_1 to the set of symbols in L_2 such that for any set of sentences Γ from L_1, the set of models $\mathcal{M}_1$ satisfying Γ and the set of models $\mathcal{M}_2$ satisfying $\psi(\Gamma)$ are equivalent modulo ω, i.e. $\mathcal{M}_1 =^{\omega} \mathcal{M}_2$.

Although the internal structures of the models for planning languages and knowledge representation languages are different, the same definition of expressivity can be used by providing a definition for equivalence of HTN models:

Definition 35 Given two HTN models M_1, M_2 for two HTN languages $\mathcal{L}_1, \mathcal{L}_2$, and a symbol translation function ω from $\mathcal{L}_1$ to $\mathcal{L}_2$, $M_1 =^{\omega} M_2$ iff

1. For any ground primitive task α, any state s, and any ground literal l in $\mathcal{L}_1$, l is true in $\mathcal{F}_{M1}(\alpha, s)$ iff $\omega(l)$ is true in $\mathcal{F}_{M2}(\omega(\alpha), \omega(s))$.

2. For any ground non-primitive task α in $\mathcal{L}_1$,
   $\omega(\mathcal{T}_{M1}(\alpha))$ covers $\mathcal{T}_{M2}(\omega(\alpha))$, and $\mathcal{T}_{M2}(\omega(\alpha))$ covers $\omega(\mathcal{T}_{M1}(\alpha))$.

This definition states that when two HTN models are equivalent (modulo a symbol translation function) then each primitive task symbol is interpreted to have the same effects regardless of the model, and each compound task symbol is mapped to an equivalent set of primitive task networks in each model. Recall that "covers" was defined in Section 3.3.

Model-theoretic expressivity for planning languages can be formally defined as follows:

Definition 36 (Model-theoretic Expressivity) A planning language $\mathcal{L}_1$ can be expressed by a planning language $\mathcal{L}_2$, iff the following three conditions hold:

1. There exists a symbol translation function ω from the set of symbols in $\mathcal{L}_1$ to the set of symbols in $\mathcal{L}_2$,

2. There exists a function ψ from the set of sentences in $\mathcal{L}_1$ to the set of sentences in $\mathcal{L}_2$,

3. For any set of sentences Γ_1 in $\mathcal{L}_1$, whenever $\mathcal{M}_1$ is the set of all models that satisfy Γ_1 and $\mathcal{M}_2$ is the set of all models that satisfy $\psi(\Gamma_2)$, it is the case that $\mathcal{M}_1 =^{\omega} \mathcal{M}_2$.

Thus, a planning language $\mathcal{L}_1$ can be expressed by a planning language $\mathcal{L}_2$, if for any set of sentences in $\mathcal{L}_1$, there is a corresponding set of sentences in $\mathcal{L}_2$ with the same meaning (according to model-theoretic semantics).

When a planning language $\mathcal{L}_1$ can be expressed by a planning language $\mathcal{L}_2$, but not vice versa, then $\mathcal{L}_2$ is strictly more expressive than $\mathcal{L}_1$.

Note that, unlike complexity-based expressivity, model-theoretic expressivity does not put any restrictions on the computational cost of ψ, or on the size of the output of ψ in terms of its input size. This yields to a stronger definition of expressivity. If $\mathcal{L}_1$ is more expressive than $\mathcal{L}_2$ according to the model-theoretic definition, it means there are concepts representable in $\mathcal{L}_1$ but not it $\mathcal{L}_2$; it is not merely that the translation is computationally expensive or unknown.

5.1.3 Operational Expressivity

One can argue that the meaning of a set of sentences in a language is captured better by the set of *minimal* models, rather than the set of all models that satisfy that set of sentences. After all, deductions are usually based on the set of minimal models; so is the set of solutions to planning problems represented in HTN language. The definition for model-theoretic expressivity can easily be modified to accommodate that:

Definition 37 (Operational Expressivity) A planning language $\mathcal{L}_1$ can be expressed by a planning language $\mathcal{L}_2$, iff the following three conditions hold:

1. There exists a symbol translation function ω from the set of symbols in $\mathcal{L}_1$ to the set of symbols in $\mathcal{L}_2$,
2. There exists a function ψ from the set of sentences in $\mathcal{L}_1$ to the set of sentences in $\mathcal{L}_2$,
3. For any set of sentences Γ_1 in $\mathcal{L}_1$, whenever M_1 is the minimum model that satisfies Γ_1, and M_2 is the minimum model that satisfies $\psi(\Gamma_1)$, it is the case that $M_1 =^{\omega} M_2$.[1]

Note that item 3 in the definition refers to the minimum model, rather than the set of all models, as in the definition of model-theoretic expressivity.

From the operational semantics point of view, this definition requires that for every planning problem expressed in L_1, there exists a corresponding planning problem expressed in L_2 and the set of solutions for both these problems are equivalent.

5.2 Expressivity: HTNs versus STRIPS Representation

Although HTNs are more flexible compared to STRIPS-style planning, it was generally believed that anything that can be done in HTN planning can be also done in STRIPS-style planning. Due to the lack of a formalism for HTN planning, such claims could not be proved or disproved until now.

A comparison of HTN planning with STRIPS-style planning reveals that the HTN approach provides all the concepts (states, actions, goals) that STRIPS has. Conversely, STRIPS-style planning lacks the concepts of constraints, compound tasks, task networks and task decomposition. Thus, HTN planning must be more expressive. For example, consider a compound task of making a round trip to New York. This task cannot easily be expressed in state-based planing because the initial and final states would be the same. This intuition is going to be substantiated in the following sections.

5.2.1 Transformations from STRIPS to HTNs

Any planning domain encoded as a set of STRIPS operators can be transformed into an HTN planning domain, in low-order polynomial time. Figure 5.2 shows one such transformation. This transformation follows the intuition that in order to make a literal true, one chooses an operator that has the desired effect, achieves the preconditions of that operator, and then executes it, as depicted in Figure 5.1. The HTN representation for the problem requires as many operators as its STRIPS counterpart. In addition, it requires as many methods as there are effects of operators. Thus the transformation is polynomial.

[1]This definition refers to *the minimum model* instead of *minimal models* because in the HTN language the minimum model is unique modulo symbol names and "covering", as described in Section 3.2

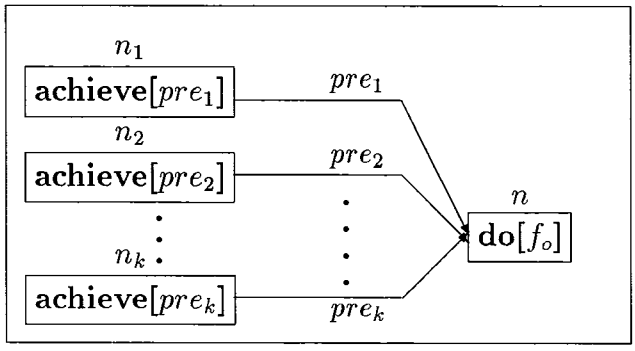


Figure 5.1: Graphical representation of a method for Transformation 1.

Transformation 1

1. Input a STRIPS problem $< O, I, G >$, where O is a set of STRIPS operators, I, G are sets of atoms representing the initial state and the goal.
2. For each operator $o \in O$, declare a primitive task f_o with the same preconditions and effects as o.
3. For each effect l of each operator
$o = [PRE:\ pre_1, pre_2, \ldots, pre_k][POST:\ l_1, l_2, \ldots, l_r]$ declare a method as depicted above in Figure 5.1:
$(method\ l$
$\quad [(n1 : achieve[pre_1]) \ldots, (n_k : achieve[pre_k])(n : do[f_o]),$
$\quad (ord\ n1\ n) \wedge \ldots (ord\ n_k\ n) \wedge$
$(between\ pre_1\ n1\ n) \wedge \ldots (between\ pre_k\ n_k\ n)]$
4. Declare I as the initial state.
5. Let $G = g_1 \wedge, \ldots, g_k$. Declare input task network d as
$\quad [(n1 : achieve[g_1]) \ldots, (n_k : achieve[g_k])(n : do[dummy]),$
$\quad (ord\ n1\ n) \wedge \ldots (ord\ n_k\ n) \wedge (between\ g_1\ n1\ n) \wedge \ldots (between\ g_k\ n_k\ n)])$

Figure 5.2: A transformation from STRIPS-style planning to HTN planning

Transformation 2

1. Input a STRIPS problem $< O, I, G >$, where $O = \{o_1, \ldots, o_k\}$ is a set of STRIPS operators, I, G are sets of atoms representing the initial state and the goal.
2. For each operator $o \in O$, declare a primitive task f_o with the same preconditions and effects as o.
3. Declare another primitive task *nothing* with no preconditions or effects.
4. Declare a compound task t.
5. Declare $(method\ t\ \ [(n:\ do[nothing])\ TRUE])$
6. For each primitive task f_o declare
   $(method\ t\ \ [(n1:\ do[f_o])(n_2:\ perform[t])\ \ (ord\ n_1\ n_2)])$
7. Let $G = g_1 \wedge, \ldots, g_k$. Declare input task network d as
   $[(n : perform[t]) \qquad (after\ g_1\ n) \wedge (after\ g_2\ n) \wedge \ldots (after\ g_k\ n)]$

Figure 5.3: Another transformation from STRIPS-style planning to HTN planning.

$$
\begin{aligned}
t &\rightarrow \quad \epsilon \\
t &\rightarrow \quad f_{o1}\, t \\
t &\rightarrow \quad f_{o2}\, t \\
&\vdots \\
t &\rightarrow \quad f_{ok}\, t.
\end{aligned}
$$

Figure 5.4: Form of Methods in Transformation 2.

Note that this transformation will exclude solutions that contain redundant actions (i.e. actions that do not assert a goal or a precondition of some subsequent action). All STRIPS-style planning systems that do precondition chaining (e.g. TWEAK [15], SNLP [46]) exclude such solutions. Nonetheless, it is possible to define polynomial transformations such that any sequence of actions for which the final state satisfies the goal is a valid solution. Figure 5.3 shows one such transformation.

This transformation introduces a compound task t. A close examination of Steps 5 and 6 in Figure 5.3 reveals that the methods for t look like context-free grammar rules as depicted in Figure 5.4.

Hence t can be expanded into any sequence of actions, provided that the preconditions are satisfied in the intermediate states and the goal is satisfied in the final state.

Note that this transformation produces only regular HTN problems, which has exactly the same complexity as STRIPS-style planning. Thus, just as restricting context-free grammars

to be right linear produces regular sets, restricting HTN methods to be regular produces STRIPS-style planning.

5.2.2 Complexity-based Expressivity Results

As a consequence of the polynomial transformations from STRIPS-style planning to HTN planning presented in the previous section, the following lemma can be stated:

Lemma 1 *There exists a polynomial transformation ψ from the set of STRIPS-style planning problem instances to the set of HTN planning problem instances such that for any STRIPS-style planning problem instance* $\mathbf{P}$, $\mathbf{P}$ *has a solution iff* $\psi(\mathbf{P})$ *also has a solution.*

The next question is whether there exists a transformation in the other direction, that is whether it is possible to encode HTN planning problems as STRIPS-style planning problems. From Theorem 4, HTN planning with no function symbols (and thus only finitely many ground terms) is semi-decidable. Even if the domain description $\mathcal{D}$ is required to be fixed in advance (i.e., not part of the input), Theorem 5 states that there are HTN planning domains for which planning is semi-decidable. However, with no function symbols, STRIPS-style planning is decidable, regardless of whether or not the planning domain[2] is fixed in advance [22]. Thus:

Lemma 2 *There does not exist a computable function ψ from the set of HTN planning problem instances to the set of STRIPS-style planning problem instances such that for any HTN-planning problem instance* $\mathbf{P}$, $\mathbf{P}$ *has a solution iff* $\psi(\mathbf{P})$ *also has a solution.*[3]

From the definition of complexity-based expressivity and Lemmas 1 and 2, it immediately follows that

Theorem 12 *The HTN language is strictly more expressive than the STRIPS language with respect to complexity-based expressivity.*

Showing whether a polynomial or computable transformation exists is one way of comparing the expressivity of two languages. The lack of a computable transformation from HTN planning to STRIPS-style planning means that for some planning problems, the problem representation in STRIPS-style planning will be exponentially (or in some cases, even infinitely!) larger than in HTN planning.

5.2.3 Model-Theoretic Expressivity Results

Before proceeding to compare HTN planning and STRIPS-style planning according to model-theoretic expressivity, a formal semantics for STRIPS-style planning that is compatible with the semantics for HTN planning is required.

[2]Since STRIPS-style planning does not include methods, a STRIPS-style planning domain is simply a set of operators.

[3]This lemma uses the standard assumption that the STRIPS operators do not contain function symbols, nor do the HTN operators.

Semantics for STRIPS

A semantic structure for STRIPS-style planning has the same form as a semantic structure for HTN planning, with some restrictions. Thus it is a triple $M = \langle \mathcal{S}_M, \mathcal{F}_M, \mathcal{T}_M \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{F}$ interprets actions as state transitions, and $\mathcal{T}$ interprets non-primitive tasks as sets of ground primitive task networks, with the following restrictions:

1. Since STRIPS representation lacks the notion of compound tasks, non-primitive tasks consist of only goal tasks, and $\mathcal{T}$ is not defined for compound tasks.

2. In STRIPS-style planning, any executable sequence of actions that make the goals true is a valid plan, thus, given any goal task $achieve[l]$, $\mathcal{T}$ maps $achieve[l]$ to the set of all ground sequences of actions, each with the (implicit) constraint that l is true in the final state.

It is fairly easy to show that the STRIPS language can be expressed by the HTN language. It suffices to present two functions ψ and ω such that the corresponding translation preserves the set of models.

Since the HTN language does not require any extra symbols for expressing the STRIPS language, the symbol translation function ω is defined to be the identity function. Thus the HTN representation is going to use exactly the same set of constants, predicates, and actions (primitive tasks) as its STRIPS counterpart.

Given a set of STRIPS-style operators Γ, here is how we define $\psi(\Gamma)$:

- $\psi(\Gamma)$ contains exactly the same set of operators as Γ.

- For each goal task $achieve[l]$ and each action f, $\psi(\Gamma)$ contains the method

$$(achieve[l], [(n_1 : do[f])(n_2 : achieve[l]), (n_1 \prec n_2)])$$

Note that the construction of ψ is very similar to the methods defined for Transformation 2 shown in Figure 5.3.

Because there is an implicit method for each goal task stating it can be expanded to a dummy task when the goal literal is already true, with the methods in $\psi(\Gamma)$, any goal task can be expanded to any sequence of actions. Such a sequence of actions would be a plan for the goal task whenever all the actions are executable and the goal literal is true in the end of the plan. Thus, the methods precisely reflect the restrictions on the models of STRIPS-style planning, and as a result Γ and $\psi(\Gamma)$ have exactly the same set of models. Thus

Lemma 3 *The STRIPS language can be expressed by the HTN language with respect to model-theoretic expressivity.*

The converse is not true. To prove that the STRIPS language is not as expressive as the HTN language, I am going to construct an HTN planning domain Γ, and show that there

does not exist any STRIPS-style planning domain (i.e. set of STRIPS operators) with an equivalent set of models.

Note that the set of plans for any STRIPS-style planning domain always forms a regular set: One can define a finite automata with the same states as the STRIPS domain, with state transitions corresponding to the actions, and the goal states in the STRIPS domain would be designated as the final states in the automata. On the other hand, the set of plans for an HTN planning domain can be any arbitrary context-free set, including those context-free sets that are not regular. Given any context-free grammar, one can declare one compound task symbol for each non-terminal of the grammar, one primitive task symbol for each terminal of the grammar, and for each grammar rule of the form $X \rightarrow YZ$, one can declare a method of the form $(\alpha_X \ [(n_1 : \alpha_Y)(n_2 : \alpha_Z) \ (n_1 \prec n_2)])$.

Given an HTN planning domain Γ that corresponds to a context-free but not regular grammar, Γ will have a minimum model[4] M such that $\mathcal{T}_M$ will map compound task symbols (which correspond to the non-terminal symbols of the grammar) to sets of totally ordered primitive task networks (or equivalently, to context-free sets of strings from the terminal symbols of the grammar). Since $\mathcal{T}_M$ maps compound tasks into context-free but not regular sets, no STRIPS-style planning domain can have a model equivalent to M. Thus

Lemma 4 *The HTN language cannot be expressed by the STRIPS language with respect to model-theoretic expressivity.*

Thus, from Lemma 3 and Lemma 4

Theorem 13 *The HTN language is strictly more expressive than the STRIPS language with respect to model-theoretic expressivity.*

5.2.4 Operational Expressivity Results

Although the definition of expressivity based on operational semantics is quite different from the definition of model-theoretic expressivity, it yields the same result: Since the set of solutions to an HTN planning problem can be any context-free set, whereas the set of solutions to a STRIPS-style planning problem always is a regular set, a proof similar to that of Lemma 4 can be constructed to prove that HTN planning is strictly more expressive according to this definition of expressivity. Thus,

Theorem 14 *HTN planning is strictly more expressive than STRIPS-style planning according to the definition of operational expressivity.*

5.2.5 Discussion of Expressivity Results

The complexity-based expressivity results has shown that HTN constructs can represent computationally more expensive problems than can be represented with STRIPS-operators.

[4]refer to the proof of Theorem 1 to see how to construct a minimum model

The modal-theoretic and operational expressivity results have shown that this is not only due to the conciseness of the HTN languages but also due to the richness of the semantic structure and the structure of the solutions to HTN planning problems.

The expressive power of HTN planning stems from two constructs:

1. Planning problems are in the form of task networks, rather than conjunctive attainment goals,

2. Compound tasks specified by methods can represent complex planning jobs.

Task networks contain multiple tasks and a constraint formula, which makes it easy to represent complex scheduling problems and provides flexibility—but if all tasks were either primitive or goal (STRIPS-style) tasks, these could probably be expressed with STRIPS-stye operators (albeit clumsily and using an exponential number of operators/predicates). On the other hand, compound tasks and methods provide an abstract representation for sets of primitive task networks, similar to the way non-terminal symbols provide an abstract representation for sets of strings in context-free grammars. This allows representing planning jobs that are too complicated to be encoded in STRIPS-representation.

Chapter 6

Algorithms used by the UMCP System

In Section 3.5, I have presented a provably correct HTN planning algorithm called UMCP. This chapter contains the elaboration of this abstract algorithm into the UMCP planning system. Section 6.1 presents the high-level search algorithm and a conceptual view of the data structures. Section 6.2 contains task expansion as a refinement method. Section 6.3 describes the constraint refinement algorithms, which function as the domain-independent critics of the UMCP system, and Section 6.4 explains how procedural domain information can be incorporated into UMCP as domain-specific critics. Finally, Section 6.7 provides a global perspective to the algorithms.

The work described in this chapter is based on the formal framework presented in Chapter 3, which provides a characterization of the set of solutions to a given HTN planning problem. The complexity analysis of HTN planning presented in Chapters 4 has influenced the design of these algorithms by pointing to the computational difficulties and bottlenecks involved in HTN planning. These algorithms have also benefited from the expressivity results in Chapter 5, which uncovered the similarities and differences between STRIPS-style planning and HTN planning. As a result, these algorithms could draw upon the huge body of work on STRIPS-style planning, and contribute back to it.

6.1 Refinement Search in UMCP

One way of finding solutions to HTN planning problems is to generate all possible expansions of the input task network to primitive task networks, then generate all possible ground instances (assignment of constants to variables) and total orderings of those primitive task networks and finally output those whose constraint formulae evaluate to true. However, considering the size of the search space, it is more appropriate to try to take advantage of the structure of the problem, and prune large chunks of the search space by eliminating in advance some of the variable bindings, orderings or methods that would lead to dead-ends. To accomplish this UMCP uses a branch-and-bound approach [39].

1. Input a planning problem $\mathbf{P} =< d, I, \mathcal{D} >$.
2. Initialize OPEN-LIST to contain only d.
3. If OPEN-LIST is empty, then
   halt and return "NO SOLUTION."
4. Pick and remove a task network tn from OPEN-LIST.
5. If tn is completely refined into a solution then
   halt and return tn.
6. Pick a refinement strategy R for tn .
7. Apply R to tn and insert the resulting set of task networks into OPEN-LIST.
8. Go to step 3.

Figure 6.1: High-level Refinement-Search in UMCP

A task network can be thought of as an implicit representation for the set of solutions for that task network. UMCP works by refining a task network into a set of task networks, whose sets of solutions together make up the set of solutions for the original task network. Those task networks whose set of solutions are determined to be empty are filtered out. In this aspect, UMCP nicely fits into the general refinement search framework described in [37].

Figure 6.1 contains a sketch of the high-level search algorithm of the UMCP system. This algorithm is a deterministic version of the UMCP algorithm presented in Section 3.5. Search is implemented by keeping an OPEN-LIST of task networks that are to be explored. Step 5 checks whether tn is a solution node: if all tasks in tn are primitive, the constraint formula consists of the atom TRUE, and the list of constraints that have been committed to be made true but not yet made true is empty, then tn qualifies as a solution node.[1] All task orderings and variable assignments consistent with the auxiliary data structures associated with tn are plans for the original problem, and these plans can be easily enumerated. If tn is *not* a solution node, then it is refined by some refinement strategy R, and the resulting task networks are inserted back into the OPEN-LIST. Note that a refinement strategy may return an empty set of task networks, which means the given task network has no solution, and thus it is pruned from the search space.

The non-deterministic UMCP algorithm described in Section 3.5 had steps for task expansion, applying critics, and also for finding completions. The deterministic version covers all of these steps as refinement strategies. The three types of refinement strategies used in UMCP are task expansion, constraint refinement, and user-specific critics.

Task expansion involves retrieving the set of methods associated with a non-primitive task in tn, expanding tn by applying each method to the chosen task and returning the resulting set of task networks.

[1] Constraints and the data structures will be discussed in detail shortly.

User-specific critics is one of the places where UMCP can be tailored for specific domains by providing a domain-specific refinement strategy. UMCP can work without domain specific critics, but if such a critic is available, it can be used to improve the performance of the planner.

Constraint refinement is UMCP's way of implementing domain-independent critics. It is one of the most important and the most complicated parts of the UMCP algorithms.

6.1.1 Properties of Refinement in UMCP

All refinement strategies in UMCP are designed to preserve soundness, completeness, and systematicity. The following is a list of conditions that need to be satisfied to preserve those properties:

- Soundness: Any solution to any task network in $R(tn)$ is also a solution for tn. Thus refinement does not introduce invalid solutions.
- Completeness: Any solution for tn is also a solution for some task network in $R(tn)$. Thus refinement does not eliminate any valid solutions.
- Systematicity: The set of candidate solutions for each task network in $R(tn)$ are mutually disjoint. Thus UMCP does not examine any candidate multiple times.

Of those three properties, soundness is the most important for most planning applications, where the correctness of plans is critical.

Completeness means the planner can always find any solution that exists. Completeness might be critical in cases where any solution, even a slightly incorrect solution is preferable to no solution at all. Completeness may also affect the performance of a planner: An incomplete planner may miss a simple solution and spend more time trying to find a complex one.

The utility of systematicity is more debatable. Systematicity avoids redundancy in the search space, thus it was believed to improve the performance of planners; however, as discussed in [35], enforcing systematicity may incur extra overhead in book keeping, which can offset the reduction in the size of the search space. Enforcing systematicity in the refinement algorithms used by UMCP does not seem to require significant overhead, but it is a matter of further investigation and experimentation to verify this opinion.

6.1.2 High-level Data Structures for UMCP

This section provides a conceptual overview of the high-level data structures used by UMCP.

6.1.2.1 OPEN-LIST

UMCP uses OPEN-LIST for its high-level search, which is a well known data structure in the search literature [39]. Each new generated node in the search space is inserted to OPEN-LIST, until it is removed for further exploration. Several search strategies can be used in

deciding which node to pick and remove from OPEN-LIST at each iteration of the search. The internal implementation of OPEN-LIST depends on this strategy. UMCP currently provides the following search strategies:

- **Breadth-first Search:** The nodes are removed from the OPEN-LIST in the order they are inserted: first in, first out.
- **Depth-first Search:** The nodes are removed from the OPEN-LIST in the reverse order they are inserted: last in, first out.
- **Best-first Search:** The nodes in the OPEN-LIST are ranked using a heuristic evaluation function, and the one with the lowest rank is removed first. The evaluation function is provided by the user; it contains domain-specific information to guide the search. If for each node in the search space, the evaluation function always returns a value less than or equal to the cost of the best solution that can be obtained from that node, then UMCP will find the optimum solution. It is possible to define evaluation functions based on the domain-independent features of the nodes (e.g., number of non-primitive tasks).

6.1.2.2 Search Nodes

Each node in the search space is in the form of a task network, with additional data structures to store plan decisions.

The three types of decisions in HTN planning are the choice of method for each non-primitive task, the choice of constant to assign to each variable, and the orderings of tasks. Of those three, the choice of method is directly reflected in the task network. When a task in a task network is expanded using a given method, the tasks in the method are added to the task list, and the constraint formula of the method is combined with the constraint formula of the original task network with a conjunct. Auxiliary data structures are required for recording the ordering and variable binding decisions. Thus, along with each task network, UMCP keeps a list of possible values for each variable, and a partial order graph of task nodes. Both of those structures will be referred to as *commitment data structures*, and the act of restricting the possible value for a given variable, or restricting the possible orderings of the tasks will be referred to as *making commitments*, or simply *commitments*.

Dealing with some constraints might not be possible at the current level of detail in a task network. For example, consider a planning problem that involves travelling. John needs to be in New York City by 5p.m. Until the task network representing the planning problem is sufficiently refined, that is, the decisions regarding the mode of transportation (fly, drive, take the train, etc.) and which flight or train to take are made, it is not possible for the planner to enforce that John is indeed going to be in New York City by 5p.m., simply by making more variable binding and ordering commitments. In those situations where it is not possible to enforce a given constraint to become true by making more commitments, the constraint is recorded in a list called the Promissory List (the list of constraints the planner has committed to make true, but has not done so yet), until such time that the task network

is sufficiently refined to enforce the constraint via further commitments. The constraints in the promissory list do not simply lie dormant until they are ripe for processing. They influence the planning decisions. For instance, if the time schedule is tight, the planner might choose flying over taking the train, although it is more expensive.

Each search node contains the following data:

- **Task list:** This is a list storing the tasks in the task network.
- **Constraint formula:** It stores the constraint formula of the task network.
- **Partial-order Graph:** It stores the ordering decisions on the tasks.
- **Possible-value list:** This list has an entry for each variable, containing the set of possible values (i.e. constant symbols) it can be instantiated to.
- **Promissory List:** This is the list of constraints the planner has committed to make true in this node, but these constraints are not enforced yet, via the commitments recorded in possible-value list and partial-order graph.

6.2 Task Expansion as Refinement

Task expansion involves retrieving the set of methods associated with a non-primitive task α in tn, expanding tn by applying each method to the chosen task α and returning the resulting set of task networks.

The set of task networks generated from task expansion was defined to be $red(tn, I, \mathcal{D})$ in the HTN formalism in Section 3.2, where I is the initial state, and $\mathcal{D}$ is the domain description (i.e., the set of methods and the set of operators in the planning domain). Section 3.2 has also precisely defined how to expand a task in a task network tn using a method m, denoted by $reduce(tn, n, m)$, where n is the label of the task being expanded.

The definition of $reduce(tn, n, m)$ is used to implement the task expansions. The implementation has additional steps to set the auxiliary data structures (i.e., the partial-order graph, the possible value list, and the promissory list) of the newly generated task networks. The values of the auxiliary data structures are copied from tn to the new task networks, and then modified as follows:

- Partial-order graph: If a task was ordered before (respectively after) the expanded task in the original task network, it must be ordered before (respectively after) all the subtasks in the expansion.
- Possible-value list: An entry must be made in the possible-value list for each new variable introduced by the expansion. Each new variable must have the set of all constant symbols as the initial set of possible values.

- Promissory list: Section 3.2 describes how to modify a constraint that refers to the label of the expanded task. These modifications, which are used for the constraint formula, must also be applied to all the constraints in the promissory list.

The above description of task expansion strictly follows the semantics of HTN planning, and thus preserves soundness and completeness. It also preserves systematicity to the extend that if there are multiple ways to generate the same task network using a different set of methods in expansions, then the generated task networks are considered distinct.

For example, if the same method is declared multiple times for a given task, then the list of task networks generated by task expansion will have overlapping solutions. Although it is possible to check whether two methods are equivalent modulo the names for variables and node labels, it is computationally rather hard even to tell whether the two lists of tasks are equivalent, let alone their constraint formulae. Even if no task has two equivalent methods, it might still be possible to generate the same task network using different methods.

6.3 Constraint Refinement in UMCP

This section presents the constraint handling mechanisms in the UMCP system. These mechanisms serve as domain independent critics, and they are designed to preserve soundness, completeness, and systematicity.

As was explained in Section 2.2.7, HTN planning systems [59, 64, 69] have used a mechanism called critics to deal with interactions. Critics helped prune the search space by detecting dead ends in advance and by resolving many types of conflicts as soon as they appeared. Due to the procedural, heuristic nature of critics, no guarantees could be made on their correctness or performance. In fact there was no criterion of correctness that could be used to evaluate a critic function.

The formal framework developed in Chapter 3 has provided a criterion for evaluating the correctness of critics. Section 3.5 presents three conditions a critic function must satisfy in order to preserve soundness, completeness, and systematicity, respectively. That Section also contains a provably correct planning algorithm UMCP (shown in Figure 3.7). The UMCP algorithm calls a critic function (Steps 6 and 7) at every iteration, with the assumption that the critic function will satisfy the conditions for correctness. This section deals with ways of implementing the critic function in such a way to satisfy the correctness conditions.

My approach to handling interactions in HTN planning is based on an insight given by the complexity analysis presented in Chapter 4: Each task in a task network introduces a number of primitive tasks as its descendants through expansions. Each task also introduces a number of constraints through the methods used in those expansions. The interaction among two tasks in a task network occur either because the constraints associated with one task conflicts with the constraints of the other task, or a descendant primitive task of one task has an effect that conflicts with the constraints associated with the other task. In brief, the interactions are based on constraints and the influence of primitive tasks on constraints. Thus interaction detection and conflict resolution can be viewed as a problem of constraint

satisfaction.

The following sections describe in detail how UMCP deals with constraints.

6.3.1 Overview of Constraint Refinement in UMCP

Constraint satisfaction in UMCP is complicated because HTN planning involves a rather rich set of constraint types, and the constraint formula of a task network changes dynamically as tasks are expanded. To provide a general view, constraint refinement techniques of UMCP are first demonstrated on a simple problem of telling whether a given propositional formula is satisfiable. Section 6.3.1.1 presents how to solve the satisfiability problem using refinement search. Section 6.3.1.2 describes how this satisfiability algorithm can be extended to HTN planning.

6.3.1.1 An Algorithm for the Satisfiability Problem

The satisfiability problem is defined as follows:

> Given a propositional boolean formula ϕ constructed using the logical connectives $\{\wedge, \vee, \neg\}$ from a set of propositions $P = \{p_1, p_2, \ldots, p_k\}$, does there exists a truth assignment $\theta : P \rightarrow \{T, F\}$ such that ϕ evaluates to true?

The satisfiability problem can be solved using the following procedure: Each proposition has two possible values, namely true and false. We choose a proposition from the formula, and create two branches in the search tree. In one branch, the proposition is assigned value true, and in the other, the value false. In each branch the proposition can be replaced with its assignment, and the formula can be simplified accordingly. This procedure can be repeated recursively on every branch, until the formula in one branch evaluates to true, in which case the initial formula is satisfiable, or all branches terminate with value false, in which case the formula is not satisfiable.

For example, consider the boolean formula

$$F = \neg p_1 \wedge (p_2 \wedge \neg p_3 \vee p_1) \wedge (\neg p_2 \vee \neg p_3)$$

The corresponding search tree is depicted in Figure 6.2. The root of the search tree is a node containing the formula ϕ, and the truth assignment θ, which is initially empty. Children of any node in the tree can be thought of as refinements of that node. Children never modify the truth assignments of their parent node, they only add more assignments. Any truth assignment that satisfies a node's formula also satisfy that of its parent. Any truth assignment that satisfies a node's formula also satisfies the formula of exactly one child. Thus, this type of refinement preserves soundness, completeness, and systematicity.

The refinement algorithm presented in Figure 6.3 describes the procedure for generating the children of a given node in the search tree. This procedure has three phases: *selection, enforcement, simplification.* Selection refers to choosing which proposition to work with first. This choice determines the shape of the search tree, and it influences the number of nodes

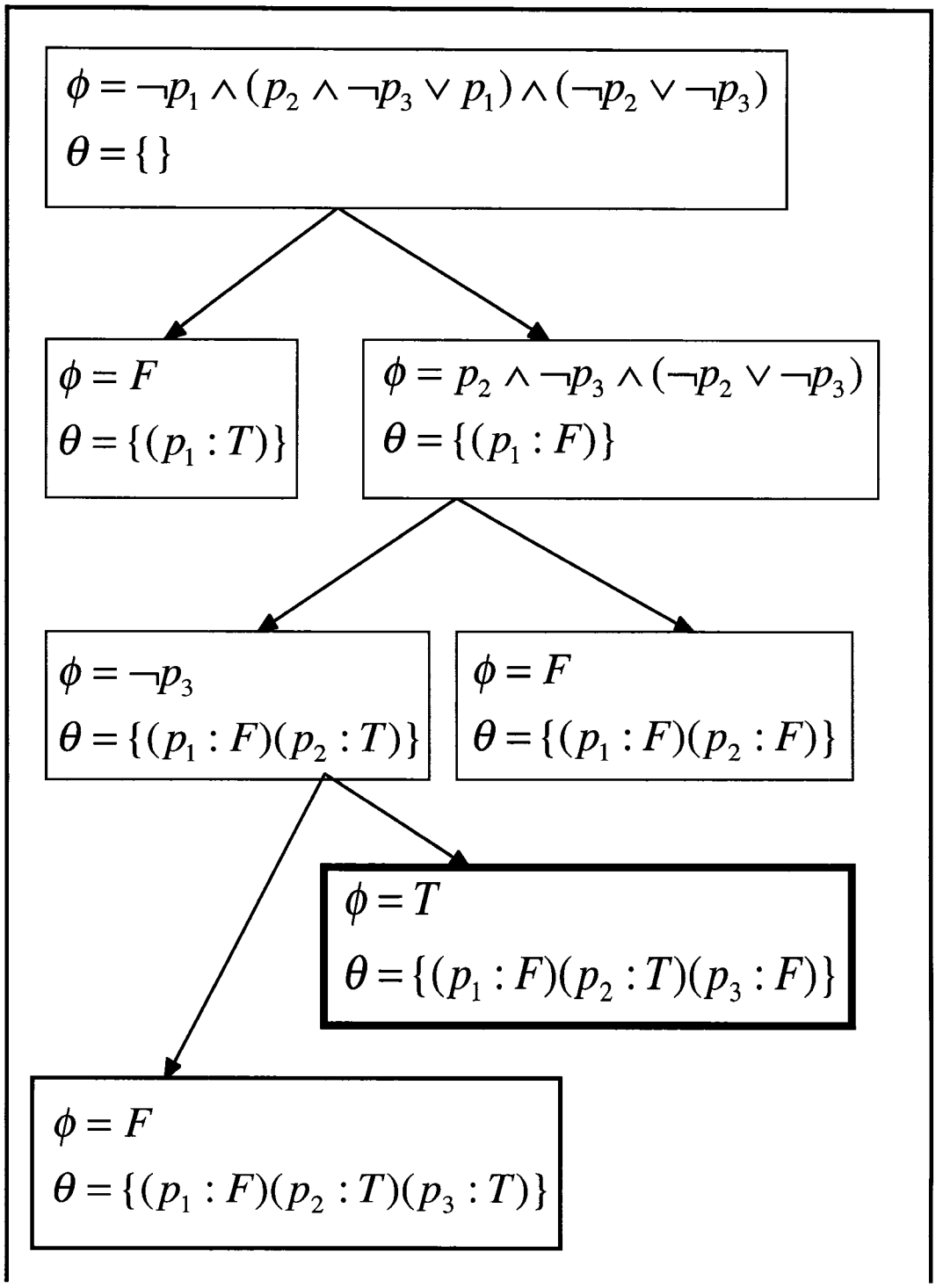


Figure 6.2: Search Tree for a Satisfiability Problem

1. Input a node $s =< \phi, \theta >$
2. [*Selection*] Pick a proposition p in ϕ
3. [*Enforcement*] Create two children nodes:
   $s_1 =< \phi, \{(p : True)\} \cup \theta >$
   $s_2 =< \phi, \{(p : False)\} \cup \theta >$
4. [*Simplification*] Simplify the constraint formula of each child node based on its truth assignment. Output those nodes whose formula does not evaluate to false.

Figure 6.3: Constraint refinement for Satisfiability Problem

that need to be explored. Enforcement refers to making further refinements to the previous decisions by recording the new truth assignment in each child node. Simplification refers to reevaluating and simplifying the formula on each branch based on the new refinements recorded in truth assignments. Those nodes whose formula evaluate to false have no solution, and thus they can be terminated.

One can think of several extensions to this algorithm. The formula may contain *unit clauses*. A unit clause is a literal (i.e. a proposition or its negation), combined with the rest of the formula with a conjunct. For instance, the literal $\neg p_1$ is a unit clause in the example above. Only one branch needs to be created for a unit clause, because enforcing it to be false will make the formula itself false. Thus it is a good idea to select unit clauses before the others. If the constraint formula contains multiple unit clauses, it makes sense to select all of them, and enforce them together to avoid doing simplification many times.

In order to support the extensions described in the paragraph above, the selection phase must return a more general form. Hence, the selection step is modified to return a list of the form

$$\begin{array}{ll} \{ & (l_{11} l_{12} \ldots, l_{1m}), \\ & (l_{21} l_{22} \ldots, l_{2m'}) \\ & \vdots \\ & (l_{k1} l_{k2} \ldots, l_{km''''}) \quad \} \end{array}$$

The enforcement phase, which receives this selection, creates k new nodes, and on i^{th} node it enforces the literals $l_{i1} l_{i2} \ldots l_{im'})$.

For example, selection may return a single list of unit clauses in the format $\{(l_1 l_2 l_3)\}$, or it may return a two lists, one list containing a single literal, the other its negation such as $\{(l_1), (\neg l_1)\}$.

The selection must have two properties:

- The selected lists of literals must be comprehensive to cover all possibilities and hence preserve completeness.
- Each list of selected literals must be mutually inconsistent in the presence of previously made decisions so that the set of solutions on each branch are mutually exclusive, and systematicity is preserved.

Soundness is preserved by definition, as previously made truth assignments are not modified in the generated nodes.

A selection of the form $\{(l_1), (\neg l_1)\}$ certainly satisfies both of the conditions above. This selection covers all possibilities, as l_1 can have only two values, either true, or false. Furthermore these two values are contradictory, so there cannot be any common solutions in the corresponding subtrees of the search space.

6.3.1.2 Phases of Constraint Refinement in UMCP

Task networks in HTN planning contain a boolean formula that must be satisfied, just like the boolean satisfiability problem discussed in the previous section.

A constraint formula in a task network is quite more complicated than a propositional formula; instead of propositions, the constraint formula is composed of several types of HTN constraints on variable bindings, task orderings and conditions on intermediate states. Furthermore, HTN constraints interact with one another. For example, making the variable binding constraint $(v = a)$ true, automatically makes $(v = b)$ false. State constraints are influenced by the ordering and effects of tasks in the task network. Nonetheless, the satisfiability algorithm in Figure 6.3 can be extended for constraint refinement in HTN planning.

The data flow of constraint refinement in UMCP is shown in Figure 6.4. The constraint selection module determines the constraints to work on. Based on this selection, the constraint enforcement module produces a set of task networks, each of which corresponds to one way of enforcing the selected constraints. The resulting task networks are piped to the constraint propagation module, which for each task network attempts to enforce the constraints in its promissory list. Those task networks are piped to the constraint simplification module, which evaluates the constraint formula of each task network, and filters the task networks whose formula evaluates to false.

The high-level algorithm for constraint refinement in UMCP is shown in Figure 6.5. This algorithm is an extension of the algorithm in Figure 6.3 for boolean satisfiability.

The following sections describes the steps of constraint refinement in UMCP.

6.3.2 Constraint Selection in UMCP

Constraint selection involves deciding which constraints to work on. Given a task network, constraint selection returns a set of constraint lists, which has the form

$$\{ \ (c_{11}c_{12}\ldots), \\ (c_{21}c_{22}\ldots)$$

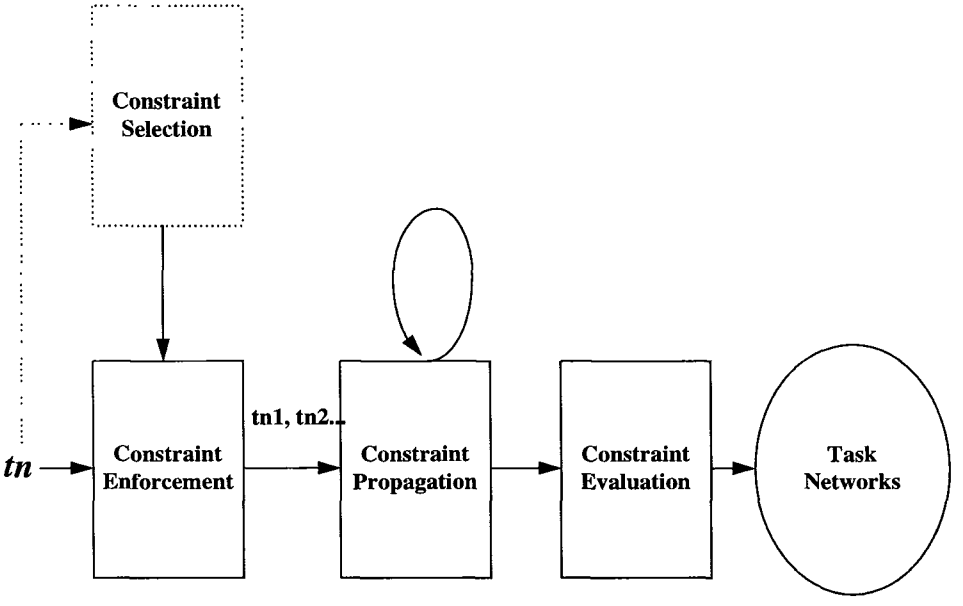


Figure 6.4: Data Flow Diagram of Constraint Refinement in UMCP

1. Input a task network tn
2. [*Constraint Selection*]
   Make a selection of the form
   $\{(c_{11}c_{12}\ldots),(c_{21}c_{22}\ldots)\ldots(c_{k1}c_{k2}\ldots)\}$
3. [*Enforcement*]
   For $i = 1$ to k do
   Produce a task network for each way of adding more commitments to tn to make $(c_{i1}c_{i2}\ldots)$ TRUE.
4. [*Propagation*]
   For each task network produced at enforcement phase do:
   Try to enforce every constraint in its promissory list to become true, until none of the constraints remaining in the promissory list can be enforced at the current level of detail in the task network.
5. [*Simplification*]
   (a) Evaluate and simplify the constraint formula of each task network produced at the propagation phase.
   (b) Output those nodes whose formula does not evaluate to false.

Figure 6.5: Constraint refinement in UMCP

$$\vdots$$
$$(c_{k1}c_{k2}\ldots) \quad \}$$

This selection tells the planning system to refine the task network into k branches. In the first branch the task network is refined further by new commitments in such a way to make all of the constraints $(c_{11}c_{12}\ldots,c_{1m})$ true. In general in branch i, the planner enforces the constraints $(c_{i1}c_{i2}\ldots)$.

The soundness of the planning system is not affected by how the selection is made. However, in order to preserve completeness, the selection must be such that any solution to the original task network lies in at least one branch, and in order to preserve systematicity, the selection must be such that any solution to the original task network lies in at most one branch. As long as these restrictions are observed, constraint selection only affects the performance of the system.

Many different strategies can be employed to make the constraint selection, and it is a matter of further experimentation to find out which provides the best performance. The strategy used by the current version of UMCP is explained below:

- If the constraint formula contains unit clauses, UMCP selects all of those. The unit clauses are ordered so that simpler types of constraints are handled before more complex ones: variable binding constraints first, initially constraints second, ordering constraints next, and the state constraints last.
- If the constraint formula does not contain unit clauses, UMCP chooses an arbitrary constraint and its negation from the constraint formula.
- If the constraint formula is TRUE, then UMCP chooses a constraint and its negation, in such a way that enforcing that constraint may simplify some of the constraints in the promissory list.
- If the constraint formula is TRUE and the promissory list is empty, then constraint refinement is not appropriate.

Note that this constraint selection strategy preserves soundness, completeness, and systematicity.

6.3.3 Constraint Enforcement in UMCP

For each type of constraint, UMCP has an algorithm to refine a task network and produce a set of task networks for ways of making that constraint true. All possible ways of making the constraint true are considered, to preserve completeness.

Details of these algorithms are described in Section 6.3.6. Constraint enforcement phase uses these algorithms to produce its set of task networks according to the constraint selection, as shown in Figure 6.6.

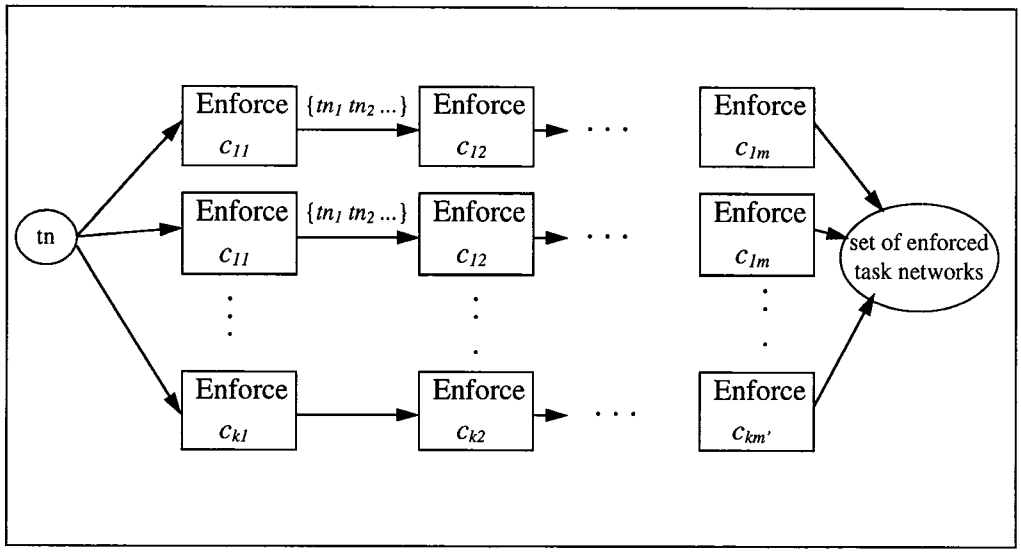


Figure 6.6: Constraint Enforcement Diagram

Suppose the constraint selection returns a set of constraint lists of the form

$$\begin{array}{ll} \{ & (c_{11}c_{12}\ldots), \\ & (c_{21}c_{22}\ldots) \\ & \vdots \\ & (c_{k1}c_{k2}\ldots) \quad \} \end{array}$$

Here is how constraint enforcement works on the first branch: It calls the enforcement function associated with the type of constraint c_{11} to produce a set of task networks for all possible ways of making c_{11} true. Each of the resulting task networks are further refined by calling the enforcement function associated with c_{12}, and so on, until finally a set of task networks are produced for each possible way of making all $(c_{11}c_{12}\ldots)$ true.

The same procedure is applied to the rest of the branches, and the resulting task networks are passed to the constraint propagation phase.

6.3.4 Constraint Propagation in UMCP

Recall the discussion in Section 6.1.2.2 about promissory list. Some constraints that go through constraint enforcement phase cannot be made true by placing restrictions on task orderings and variable bindings at the current level of detail in a task network. Those constraints are merely recorded in the promissory list. The planner makes a promise to try to establish these constraints in later iterations. Constraint propagation is the phase where this is done.

Propagation phase receives a set of task networks from the constraint enforcement phase. For each of these task networks, the promissory list is examined to see if any of the constraints in it can be enforced at the current level of detail in the task network. Those task networks with no such constraint are passed along to the simplification phase. If such a constraint is found, that the constraint enforcement function associated with the type of that constraint is performed. The task networks resulting from this constraint enforcement are piped back to the constraint propagation phase.

6.3.5 Constraint Simplification Phase

Constraint simplification is the final phase of constraint refinement. UMCP contains evaluation and simplification routines for every type of constraint, as described in the next section. These routines are used to evaluate and simplify the constraint formulae of the task networks produced by the propagation phase. For instance, if part of a conjunct evaluates to true, that part is dropped, if it evaluates to false, the whole conjunct evaluates to false. Disjuncts are treated analogously. Those task networks whose constraint formulae evaluate to false are pruned. Remaining task networks are output as the product of constraint refinement.

Sometimes it might not be possible to evaluate a constraint with the current level of detail present in a task network until further refinement. In such cases, evaluation routines of UMCP returns either the constraint itself, or a simplified version of it.

6.3.6 Details of Constraint Evaluation and Enforcement

This section describes how constraint evaluation and enforcement is done for each type of constraint. Enforcement always involves evaluating the constraint and it fails whenever the constraint evaluates to false. This is omitted from the explanations below for brevity.

6.3.6.1 Variable Binding Constraints

Recall that, each task network contains a Possible value list, which contains the set of possible values for each variable in the task network. Possible-value list is examined to evaluate variable binding constraints, and possible values for variables are further restricted to enforce variable binding constraints.

Promissory list may contain variable binding constraints, and thus it is used in the evaluation and enforcement of variable binding constraints. Only noncodesignation constraints of the form $not(v_1 = v_2)$ are stored in the promissory list. This type of constraints cannot be handled using the Possible value list, unless we instantiate one of the variables by creating a new task network for each possible value it has. Rather than instantiating one of the variables, this type of constraints are recorded in the promissory list, and they are enforced when one of the variables get instantiated later on. The remaining types of variable binding constraints can be represented using the Possible-value list, and are never recorded in the promissory list.

Details for each form of variable binding constraint appears below.

Type: $(v = a)$
Evaluation: Return true if constant a is the only possible value for variable v; return false if a is not a possible value for v; return ($v = a$) otherwise.
Update: To make it true, set the possible value list for v to a, replace v with a throughout the task network. To make it false, remove a from the possible value list for v. If v has only one possible value left, then substitute that value for v throughout the task network. If v appears in noncodesignation constraints in the promissory list, then do further refinements to enforce those constraints.

Type: $(v_1 = v_2)$
Evaluation: Return true if both v_1 and v_2 have the same one possible value; return false if they do not have any common possible values or if negation of the constraint is in the Promissory List; return the constraint itself otherwise.
Update: To make it true, set the possible value list for v_2 to the intersection of possible values for v_1 and v_2, set the possible value list for v_1 to v_2, and replace v_1 with v_2 throughout the task network. If the intersection has only one possible value, then instantiate both

variables to the value, and propagate the noncodesignation constraints in the promissory list. To make it false, insert its negation in the Promissory List.

6.3.6.2 Ordering Constraints

Ordering constraints are handled by querying and modifying the partial order graph. They are of the form (node-expression $\prec$ node-expression).

A node-expression can be in either of the two forms: $first[n_1, n_2, \ldots, n_k]$, or $last[n_1, n_2, \ldots, n_k]$. Thus, a total of four types of ordering constraints need to be considered.[2]

Let $A_i = \{n_{i1}, \ldots, n_{ik}\}$ be lists of node labels for $i = 0, 1$. Recall that $first[A_0]$ and $last[A_0]$ refer to the node which is ordered to be the first and the last among the nodes in the expansion of nodes $\{n_{01}, \ldots, n_{0k}\}$, respectively.

Let's define $O_i, I_i \subseteq A_i$ as the nodes that are not ordered after (respectively before) any node in $A_0 \cup A_1$.

Let's define $B_i, C_i \subseteq A_i$ as the nodes that are not ordered after (respectively before) *some* node in A_i and are not ordered after(respectively before) *all* nodes in $A_{(i+1) \bmod 2}$.

Type: $(first[A_0] \prec first[A_1])$
Evaluation: Return true if O_1 is empty; return false if O_0 is empty; return $(first[O_0] \prec first[O_1])$ otherwise.
Update: If O_0 contains a single primitive task put links from the node in O_0 to each node in O_1 in the partial order graph; otherwise insert $(first[O_0] \prec first[O_1])$ into the Promissory List.

Type: $(first[A_0] \prec last[A_1])$
Evaluation: Return false if B_0 or C_1 is empty; return true if some node in B_0 is ordered before one in C_1; otherwise return $(first[B_0] \prec last[C_1])$.
Update: If both B_0 and C_1 contain single primitive tasks, put a link between them, otherwise insert $(first[B_0] \prec last[C_1])$ into the Promissory List.

Evaluations and enforcements of $(last[A_0] \prec first[A_1])$ and $(last[A_0] \prec last[A_1])$ are done analogously.

6.3.6.3 State Constraints

The set of atoms in the initial state are stored in a discrimination tree for fast querying. Negative literals need not be stored due to the closed world assumption. Note that "$\neg$ (initially l)" and "(initially $\neg l$)" have the same meaning.

Type: `(initially` l`)`
Evaluation: Return true if all ground instances of l that are consistent with possible values lists in commitments are in initial state; return false if none of the ground instances of l that are consistent with possible values lists are in initial state.

[2]Ordering constraints which contain negation or refer to single node labels can be converted to one of those four forms.

Update: To make it true (false) do the following: If l has only one variable, restrict that variable to values for which l is true (false) in the initial state. If l contains $k > 1$ variables, for each combination of values for the first $k-1$ variables, output a task network by assigning those variables the corresponding constants, and restricting the value of the last variable so as to make l true (false) in the initial state. Combinations of values for which this is not possible need not be considered. For example, in order to make **(initially type[v,truck])** true in transport logistics domain, it suffices to restrict possible values for v to constants of type truck.

The state constraints of the form (l, n) are evaluated and enforced by computing the effectors. An effector of l is either (i) a primitive task with an effect that can unify with l or $\neg l$, (ii) the initial state, (iii) a compound task whose expansion might contain such a primitive task, or (iv) a committed-to state constraint that is stored in the Promissory List whose literal can unify with l or $\neg l$. Those effectors that are ordered after n or shadowed by[3] some other effector of l are ignored. If all effectors of l are positive then (l, n) is true, if all are negative than it is false, otherwise it is unknown yet. Enforcement is delayed if some of the positive effectors are compound tasks, and the constraint is recorded in the Promissory List. Otherwise for each positive effector e a new task network is created where e is the establisher of l with added constraints $(protect\ l\ e\ n) \wedge (protect\ \neg l\ e\ n)$preventing any action between e and n from denying or asserting l, respectively.

Evaluating and/or updating constraints of the form (n, l) is delayed until n refers to a single primitive task symbol. In that case (n, l) evaluates to true if the action labeled by n asserts l, it evaluates to false if that action denies l, otherwise it evaluates to (l, n).

Constraints of the form (n, l, n') are converted to $(n' \prec n) \vee [(n \preceq n') \wedge (n, l) \wedge (protect\ l\ n\ n')]$ and handled as such.

6.4 Domain-Specific Critics

Domain-specific critics is one of the places where domain-specific, procedural information can be used to enhance the capabilities and performance of UMCP.

Given a planning application, it is desirable to be able to write a specification of it using the declarative HTN planning constructs, and let an HTN planner like UMCP process the specification, and solve all the problems that arise in that application. Unfortunately, the world is seldom so well behaved.

There are types of knowledge best encoded using procedural means. Even when all the information related to a planning application can be declaratively specified, HTN planning constructs might be ill-equipped to represent and process some of this information. For example, consider planning applications involving automated manufacturing. A significant portion of the information regarding tolerances, stress analysis and many other features is very complex, and draws upon a huge body of research and experience in mechanical engi-

[3]An effector x shadows an effector y if x is ordered between y and n, and whenever an effect of y codesignates with l, so does an effect of x.

neering. Even the declarative aspects of this type of applications usually involve geometrical reasoning on solid models. On the other hand, there are many aspects of the problem that can be handled via HTN constructs, such as scheduling tools and machines, interactions among other resources and materials, choosing among process plans, etc.

One possible approach is to try to come up with more general planning paradigms that can deal with more variety of information types compared to HTN planning; however, there is a delicate balance: extending the expressive power of a declarative paradigm usually makes it computationally more expensive, sometimes to the extend of rendering it useless. In addition, there might already exist domain-specific techniques that handle many aspects of the planning application effectively.

UMCP's approach is to provide a way domain specific techniques can be incorporated to the planner in order to increase the performance and the capabilities of the system. UMCP provides an API for task networks and HTN refinement routines to facilitate integration of other code. Domain specific critics can be fully automated, or it may involve a human operator providing guidance to the system. There is no reason why other planning systems based on different paradigms cannot be used as domain-specific critics. For example, using a case based planning system that can learn how to "criticize" task networks in a given domain working in tandem with UMCP may be very beneficial.

UMCP only expects the domain-specific critics to input a task network, and output a set of task networks. It is the responsibility of the domain engineer to ensure that the domain-specific critics satisfy the conditions for preserving soundness, completeness, and systematicity.

6.5 Selection of Refinement Strategy

The previous sections have described the various refinement strategies at the disposal of UMCP. As shown in Figure 6.1, The high-level UMCP algorithm picks a refinement strategy at each iteration. This choice, which determines the part of the problem to focus on, has a big impact on the efficiency of UMCP. Intuitively, one would first like to work first on the most critical part, the part which is most likely to fail. For example, while planning for a trip, one makes the flight arrangements before planning the trip to the airport so that backtracking caused by planning for a trip to an airport with no suitable flights can be avoided.

How to choose a refinement strategy at each iteration is called *commitment stategies*, as was discussed in Section 2.2.4. UMCP supports this active area of research by providing a testbed where different commitment strategies can be implemented and compared. UMCP also provides a default commitment strategy as described below.

If the constraint formula of tn contains any unit clauses, UMCP chooses those clauses for constraint refinement, as outlined in Section 6.3.2.

If the constraint formula of tn does not contain any unit clauses, then UMCP arbitrarily picks an atomic constraint from the constraint formula and sends that constraint and its negation to constraint refinement.

If the constraint formula is true, but *tn* has non-primitive tasks, then UMCP picks one of these non-primitive tasks for task expansion. UMCP requires all the task symbols to be declared in advance, and those declarations can optionally specify a criticality level for each non-primitive task, analogous to the criticality assignments used by the ABSTRIPS system [60]. That system was described in Section 2.2.2. UMCP chooses the non-primitive task with the highest criticality level. Just as in ABSTRIPS, the criticality assignments in UMCP, provide valuable guidance in focusing the attention of the planner to the most critical part of the problem.

If the constraint formula is true and *tn* is primitive, then UMCP picks an atomic constraint (and its negation) for constraint refinement in such a way to simplify the auxiliary constraints in the promissory list.

If the promissory list is empty, the constraint formula is true, and the task network is primitive, then the task network has already been refined to a solution, and no further refinement is necessary.

Notice that the default commitment strategy of UMCP never picks domain-specific critics, as it cannot have any information about when such a critic is applicable, or how often it must be invoked. When UMCP is customized for a domain and provided with a domain-specific critic, the commitment strategy must be modified accordingly.

6.6 The UMCP Architecture

6.6.1 Modules

UMCP system has been implemented using Allegro Common Lisp on a Sparc Workstation. It consists of several modules described below.

Search module contains the high-level refinement search routine of UMCP. Depth first search, breadth-first search, and best-first search are also implemented in this module.

Heap module contains the implementation of a general purpose heap data structure (courtesy of William Andersen). It is used by the best first search module.

Refinement module contains the functions for choosing the refinement strategy to apply at a each iteration, and then invoking the appropriate module to apply the selected refinement strategy.

User-critics module is to be provided by the user to contain domain specific information. This information is used to improve the performance of UMCP.

Expansion module implements the task expansion.

Method module contains the functions for declaring, retrieving and instantiating methods.

Constraint module contains functions that implement the high-level algorithms for constraint refinement.

The following four modules contain the functions for evaluating and enforcing variable binding, ordering, initial-state constraints respectively: vb-bindings, ord-constraints, initially, and state-constraints.

State constraints are rather complicated, and they require support from several other modules described below.

Aux module contains the functions for establishing state constraints, and dealing with auxiliary data structures in the task network.

Effectors module contains the routines which compute the necessary/possible contributors or deniers to a given state constraint.

Poss-effects-table module does a preprocessing of the domain specification to determine the set of predicates a given non-primitive task may influence.

Task network module define the structure of task networks, and the basic operations on them. Several major components are delegated to other modules described below.

Formula module contains the functions on constraint formulae.

Partial-order graph module contains the functions for querying and modifying the task orderings.

Poss-values module contains the functions for querying and modifying possible values for each variable.

The promissory list, which contains the constraints that cannot be enforced right away, is implemented using two modules

Disjoint module stores the noncodesignation constraints in the promissory list.

Promissory module stores the state constraints.

Symbol module contains the functions for defining the HTN language for an application. Thus it contains the functions for declaring predicates, variables, and other types of symbols.

Util module contains miscellaneous support functions.

6.6.2 User Interface

The planner is invoked using the command `(search-for-plan <tn>)`. At each iteration, UMCP picks a task network from the openlist, chooses a refinement strategy, does the refinement, and inserts the resulting task networks back to the openlist.

UMCP presents the task network to be refined to the user as follows:

A partial order graph is visually displayed to present the list of tasks, and also the committed orderings. Primitive tasks are written in lowercase letters, and non-primitive tasks are written in uppercase.

Remaining components of the task network are displayed textually:

- Name: UMCP names each task network it generates, by extending the name of the parent task network with a "-" and the order of the task network among its siblings.
- Formula: is the constraint formula of the task network.
- Variables: presents the set of possible values for each variable. It is in the form of an association list, where the keys are the variables in the task network.
- Disjoint variables: is an association list that for each variable contains the list of variables it must noncodesignate.

- Postponed ordering constraints: is a list of ordering constraints in the promissory list.
- Postponed state constraints: is an association list of state constraints in the promissory list. It is indexed by the predicates.
- Refinement Strategy: shows which refinement strategy UMCP has chosen to refine the current task network.

The user may override UMCP's choice of refinement strategy and pick another one, and instruct UMCP to refine the current task network. Alternatively, the user can give the "go" command telling UMCP to run to completion.

6.7 Discussion

Causal links are used by POCL planners such as SNLP [46] to establish preconditions and to detect threats. Causal links are also employed by UMCP in the form of special state constraints stored in the Promissory List. SNLP's threat removal process is similar to how UMCP handles those special constraints in its constraint propagation phase.

[15] introduced the MTC (modal truth criterion) to tell whether a literal is true at a given point in a partially-ordered plan. In order to evaluate state constraints, UMCP uses an extended version of the MTC that also accounts for compound tasks. UMCP's extended MTC algorithm runs in quadratic time—and it is directly applicable for computing Chapman's MTC, for which the other known algorithms run in cubic time.

NOAH [58] employs its *resolve conflicts* critic to deal with deleted-condition interactions, which are explicitly represented by state constraints in UMCP. The constraint refinement techniques of UMCP guarantees these interactions will be handled without sacrificing soundness or completeness.

UMCP evaluates each constraint before trying to make it true, and skips those constraints that are already true, and hence it emulates NOAH's *eliminate redundant preconditions* critic.

HTN planners often allow several types of conditions in methods. How to deal with those conditions has been a topic of debate.

NONLIN [64] evaluates filter conditions as soon as they are encountered, using the QA (Question Answering) mechanism. QA returns false unless it can verify those conditions to be necessarily true, even if the conditions are possibly true. Thus, NONLIN often backtracks over filter conditions which would have been achieved by actions in later task expansions or by more ordering and variable binding commitments. As a result, NONLIN may fail to find a solution when a solution exists, or may miss a short and simple solution and do much more work to find a longer and more complicated solution.

At first glance, the problems such as these might seem to argue against the use of filter conditions: at one extreme, using filter conditions immediately to prune the search space sacrifices completeness, and at the other extreme, postponing their use until the plan is

complete (so as to preserve completeness) is inefficient.[4]

Although the above argument is partially correct, it ignores a third possibility that lies between the two extremes. In general, to preserve completeness, a planner cannot use a filter condition to prune the search space unless the filter condition evaluates to "necessarily false"—but this does not necessarily require that the task network has been expanded into a primitive and totally-ordered plan. Instead, UMCP simply records the filter conditions in the Promissory List and prunes the task network only when one of them becomes necessarily false.

More specifically, UMCP handles filter conditions and other constraints as follows:

- Some instances of variable binding, ordering, and state constraints can be dealt with immediately. For example, conditions (e.g., an object's type) that are not affected by the actions are represented by constraints of the form (**initially** l). Such constraints can be evaluated at any time by querying the initial state, and they can be committed to by appropriately restricting the possible values for the variables in l.[5]

- Those constraints that cannot be dealt with immediately are stored in the Promissory List, and are processed in the constraint propagation phases.

Constraints in UMCP go through three stages: they first appear in constraint formula; then possibly in the Promissory List if they cannot be dealt with at the time they are selected in constraint selection phase; and finally they are reflected in restrictions on possible values for variables and task orderings. This three-stage approach facilitates dealing with the disjunctions in the constraint formula, and by postponing its processing of some types of constraints, UMCP preserves completeness without sacrificing efficiency.

Dealing with numerous types of interactions is an important aspect of planning systems. The work described in Chapter 3 has provided a formal framework for representing interactions and conflicts via constraints, and in this paper we have introduced techniques for constraint handling as a means for detecting interactions and resolving conflicts. Those techniques preserve soundness, completeness, systematicity, and they have been implemented in UMCP, an HTN planning system.

By instantiating the constraint selection strategy in different ways, various commitment strategies discussed in the literature can be used by UMCP. For example, variable instantiation can be done before anything else (as in NONLIN), all primitive tasks can be totally ordered as soon as they appear in task networks, or task expansions can be deferred until all conflicts have been resolved (least commitment). Currently, we are designing experiments to empirically evaluate these techniques.

The constraint-handling mechanisms of the UMCP system provide the capabilities of many domain-independent critics discussed in the literature, and UMCP's user-specific crit-

[4]In fact, Collins and Pryor [16] have made a similar argument against filter conditions in the context of planning with STRIPS-style operators.

[5]SIPE [72] uses a "sort hierarchy" for this purpose, the only difference in UMCP is that UMCP allows arbitrary boolean formulae constructed from all types of constraints, instead of a conjunct of constraints as in SIPE.

ics module can be used to incorporate domain-specific critics as well. The modular and formal nature of UMCP makes it readily extensible. We are currently exploring ways of extending UMCP's constraint-handling mechanism to handle numerical and complex temporal constraints so that it can do deadline and resource management, and provide the capabilities of other domain-independent critics.

Chapter 7

Examples

The previous chapters have presented the HTN language, and the UMCP planning system that solves problems specified in that language. This chapter demonstrates how UMCP works on two sample domains. Section 7.1 briefly explains how to write domain specifications in the HTN language. Section 7.2 describes UM Translog, a benchmark domain for planning applications, followed by a step-by-step trace of UMCP on a sample problem from that domain. Section 7.3 presents a special planning domain, which was constructed to prove the undecidability of HTN planning in the proof of Theorem 5. In this domain, plans correspond to the intersection of the two context-free grammars, whose rules are specified in the initial state. That section also contains a sample problem trace.

7.1 Writing Domain Specifications

For most planning applications, preparing a domain specification is a challenging job. Unfortunately, it is the most neglected aspect of planning, and there is not an established software-engineering methodology to guide this job. In this section, I will attempt to provide a rudimentary approach to writing domain specifications in the HTN language. An outline of this approach is depicted in Figure 7.1. Each step is explained below, together with the UMCP syntax. UMCP uses a slight variation of the HTN language, which is easier to type on a computer terminal.

The first step in writing specifications is to identify the objects in the application. The transport logistics domain presented in Section 7.2 has several packages such as `package-1` and `package-2`, several vehicles such as `truck-1` and `airplane-4`. The domain engineer must invent names for the objects, and declare them. In the UMCP syntax, this declaration is of the form

```
(constants <constant-symbol-1> <constant-symbol-2>...)
```

The second step is to identify the relationships and properties among the objects. For example, the object `truck-2` is of type truck. At any given time, it is located *at* some

1. Identify the objects of interest in the application, and declare a constant symbol for each object.
2. Identify the properties of the objects, and the relationships among them, and declare a predicate symbol for each relation or property.
3. Identify the actions that can be executed, and declare a primitive-task symbol for each of them.
4. Identify the jobs that need to be planned for, and declare a compound-task symbol for each of them.
5. Operators and methods use variable symbols as place holders. Variable symbols must be declared as needed.
6. For each action, determine the effects and applicability conditions, and declare it in the form of an operator.
7. For each job, determine the possible ways of accomplishing it, and declare each possible way in the form of a method.
8. Prepare test problems and run them on a planning system. Based on the outcomes, revise the domain specification as necessary.

Figure 7.1: Steps of Domain Specification in the HTN Language

location. One may consider denoting these relations using two predicate symbols `truck` and `at`. In the UMCP syntax, predicates are declared using the form

```
(predicates <predicate-symbol-1> <predicate-symbol-2>...)
```

Properties of objects themselves may have relations among them. For instance, in the transport logistics domain, certain types of packages are compatible with only certain types of vehicles. Packages of type produce can be carried with only vehicles of type refrigerated. The HTN language (or any first order language) does not allow reasoning about the properties of predicates. One solution around this problem is converting some predicates into constant symbols. For example, the predicate `refrigerated` can be declared as a constant instead, and `refrigerated(truck-1)` can be represented as `type(truck-1, refrigerated)`, using a special `type` predicate. Now that vehicle types and package types are denoted by constant symbols, we can introduce a compatibility predicate `compatible(vehicle-type,package-type)`. This technique is used extensively in UM Translog.

The next two steps involve determining the actions and jobs in the domain, and declaring primitive-task symbols and compound-task symbols for them, respectively. This is accomplished using the declarations

```
(primitive-tasks <primitive-task-symbol-1>
                 <primitive-task-symbol-2>...)

(compound-tasks <compound-task-symbol-1>
                <compound-task-symbol-2>...)
```

Compound-task or predicate declarations can optionally specify a criticality level for each symbol, which UMCP uses in determining what part of a task network to work on at each iteration. The syntax for specifying criticality level in a declaration is `(<symbol> <criticality-level>)`.

Another type of symbols in the HTN language are the variables, which are used as place holders in operators and methods. Variable symbols that appear in domain and problem specifications are required to be declared in advance, using the form

```
(variables <variable-1> <variable-2>...)
```

The final two steps involve defining the primitive tasks using operators, and defining the non-primitive tasks using methods. Primitive tasks are defined in terms of their effects and their applicability conditions using the format

```
(operator <primitive-task-symbol>
 :pre ( <literal-1> <literal-2>...)
 :post( <literal-1> <literal-2>...)
)
```

A literal is written as

```
(<predicate-symbol> <term-1> <term-2> ...)
```

Terms are either constant or variable symbols. For example, the `load-package` primitive task in the UM Translog domain can be declared as follows:

```
(variables p v l)
(operator load-package(p v l)
   :pre ( (at-package p l)  (at-vehicle v l)  )
   :post( (at-package p v)  (~at-package p l) )
)
```

Methods are declared using the following syntax:

```
(declare-method <non-primitive-task-symbol>
     :expansion
     (
       (<node-label> <task-symbol> <term-1> <term-2>...)
       (<node-label> <task-symbol> <term-1> <term-2>...)
            . . .
      )
     :formula <constraint-formula>
)
```

A task symbol can be either a primitive-task, a compound task, or a predicate (possibly negated). The syntax for the constraint formula is depicted below:

formula → T
formula → 'F
formula → atomic-constraint
formula → (not formula)
formula → (and formula-1 formula-2...)
formula → (or formula-1 formula-2...)

Below is the format of atomic constraints:

- Variable binding constraints are written as `(veq term-1 term-2)`, where each term is either a variable or a constant symbol.
- Ordering constraints are written as `(ord point-1 point-2)`. Each point is either a single node label, or in one of the forms `(first node-label node-label ...)` and `(last node-label node-label ...)`.

- Constraints of type initially are written as `(initially literal)`
- State constraints have one of the forms `(before literal point)` `(after literal point)`, `(between literal point-1 point-2)`.

The following is a sample method declaration from the UM Translog domain:

```
(variables vehicle location origin r ocity dcity)
(declare-method AT-VEHICLE(vehicle location)
   :expansion
       ( (n1  MOVE-VEHICLE vehicle  origin location r) )
   :formula
       (and
           (before (AT-VEHICLE vehicle  origin)  n1)
           (initially (TYPE vehicle TRUCK)  )
           (initially (IN-CITY origin ocity)  )
           (or
              (and ; same
                  (veq r LOCAL-ROAD-ROUTE)
                  (initially (IN-CITY location ocity)  ))
              (and ;different city
                  (initially (IN-CITY location dcity) )
                  (initially (CONNECTS r ROAD-ROUTE ocity dcity)))
       ))
)
```

This is a method for moving a vehicle to a given location. This particular method only works for vehicles of type truck, under the condition that the location is either in the same city as the vehicle, or it is in a city directly connected via a road route to the current city the vehicle is located at.

As seen in this example, it is possible to add comments to domain specifications using the LISP language syntax. The semicolon character ";" is the comment symbol. It instructs the planner to ignore the rest of the line.

7.2 Case Study 1: UM Translog Domain

7.2.1 Description

The empirical studies on planning systems and techniques have been mostly on simple toy domains. Two well known examples are "Blocks World" and "Towers of Hanoi." As planning systems grow in sophistication and capabilities, however, there is a clear need for

planning benchmarks with matching complexity to evaluate those new features and capabilities. UM Translog is a planning domain designed specifically for this purpose by Andrews, Kettler, and myself [4]. This section provides a brief description of the UM Translog domain from that paper. The full domain specification is available online at: http://www.cs.umd.edu/projects/plus/UMT.

UM Translog was inspired by the CMU Transport Logistics domain developed by Manuela Veloso [67]. UM Translog is also an abstract, toy planning domain, but compared to the CMU Transport domain, it is an order of magnitude larger in size (41 actions versus 6), number of features and types of interactions. It provides a fairly rich set of entities, attributes, actions and conditions, which can be used to specify rather complex planning problems with a variety of plan interactions.

In this domain, the planner is given one or more tasks, which typically involve the delivery of a particular package to a destination. Our goal for UM Translog domain was to create a domain more complex than toy domains such as blocks world. To do this we modelled additional aspects of transport logistics not in the CMU Transport Logistics domain, and which we believed were somewhat realistic. These include the following features and restrictions:

- transport is by air, rail, or road
- transport is intracity or intercity
- several basic methods of transport are available including local transport via road, direct transport via a specific direct route, and "indirect" transport via a transportation hub
- customer locations and transportation centers (airports and train stations) are grouped into cities which are grouped into regions
- intercity transport uses specific routes
- transport via air or rail uses specific transportation centers (airports and train stations)
- particular vehicles and packages have special (un)loading methods and actions
- packages and vehicles have (sub)types which must be compatible
- vehicles, equipment, routes, and transportation centers may be temporarily unavailable
- certain cities may not allow hazardous packages to be transported through them

7.2.1.1 Objects

UM Translog objects include individual locations (cities, etc.), routes, vehicles, equipment, and packages. Each object has a primary type, specified by the predicate `type`. Primary object types include location types, route types, vehicle types, equipment types, and package

types. Types are declared as constants rather than as predicates, because we would like to be able to assert compatibility relations among vehicle types, package types and route types.

As shown in Figure 7.2, location types are `region`, `city`, and `city-location`. City location subtypes are `tcenter` (transport center) and `not-tcenter` (a city location that is *not* a transport center). Transport center subtypes are `airport` and `train-station`. Non-transport center subtypes are `clocation` (customer location) and `post-office`.

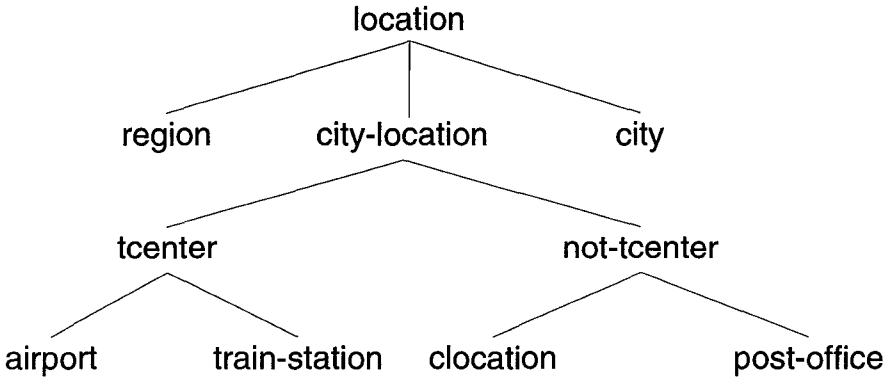


Figure 7.2: Location Type Hierarchy

Regions contain one or more cities specified via the predicate `in-region`.

Cities contain one or more city locations specified via the predicate `in-city`. Some cities are compatible with hazardous packages, specified via the predicate `pc-compatible`.

A city location is located in a specific city specified via the predicate `in-city`.

A transport center can be used for air/rail direct and indirect transport. Transport centers can optionally be specified as transport hubs via the `hub` predicate. Hub transport centers can be used for indirect transport. Transport centers serve specific cities specified via the predicate `serves`. Thus air or rail travel from a specific city must use a transport center that serves that city. Hub transport centers serve specific *regions*, rather than cities. Transport centers can be available or not. For example, a particular airport may be temporarily unavailable due to bad weather.

Customer locations are generic locations (e.g., businesses, homes, etc.) within a city that can serve as the origin and destination of a package, as can transport centers. Each customer location is located in a city specified via the predicate `in-city`.

A post office is similar to a customer location but can be used as the origin and destination for packages of type `mail`.

Route types are `road-route`, `rail-route`, and `air-route`. Routes connect locations. Road routes connect two cities. All locations within a city are assumed to be connected by roads, and thus road routes are not specified between individual city locations. Rail and air routes connect airports and train stations, respectively. Routes have an origin, destination, and route type specified via the predicate `connects`. Note that routes are directional: traffic

flows from the origin to the destination. Routes have an availability status specified via the predicate `available`. For example, a particular road route may be temporarily unavailable due to construction.

Primary vehicle types are `truck`, `airplane`, `train`, and `traincar`.

Trucks and traincars have subtypes: a single physical subtype and an optional specialty subtype. Physical truck/traincar subtypes are listed below with some examples:

- `regular`: tractor-trailer truck, delivery van, boxcar, etc.
- `flatbed`: flatbed truck, flatcar, etc.
- `tanker`: tanker truck, tanker car, etc.
- `hopper`: dump truck, hopper car, etc.
- `mail`: mail truck, mail car, etc.
- `livestock`: livestock truck, cattle car, etc.
- `auto`: car carrier truck/traincar

Specialty truck/traincar subtypes are `refrigerated` and `armored` to carry produce type of packages and valuable packages respectively. Specialty subtypes cannot be specified if the truck or traincar has a physical type of `mail`, `livestock` or `auto`. Vehicles of type `train` (i.e., train engines) unlike other types of vehicles, do not hold packages themselves but rather have attached traincars that hold packages.

A vehicle's primary type determines its compatibility with particular routes. Vehicles have a location and availability specified via the predicates `at-vehicle` and `available`, respectively.

Equipment types are `plane-ramp` and `crane`. Equipment of these types is used to load planes and flatbed trucks/traincars, respectively. Equipment has a location, specified via the predicate `at-equipment`. The status of a plane ramp is described using the predicate `ramp-connected`. The status of a crane is described using the predicate `empty`.

Packages have type `Package`. Packages have subtypes: a single physical subtype and, optionally, one or more specialty subtypes. Physical package subtypes are listed below with some examples:

- `regular` : parcels, furniture, etc.
- `bulky` : steel, lumber, etc.
- `liquid` : water, petroleum, chemicals, etc.
- `granular` : sand, ore, etc.
- `mail` : letters sent through the postal service

- `livestock` : cattle, etc.
- `auto` : automobiles

Specialty package subtypes are `perishable` for frozen food, etc.; `hazardous` for petroleum, nuclear waste, etc.; and `valuable` for money, weapons, etc. Specialty subtypes cannot be specified if the package has a physical subtype of `mail`, `livestock`, or `auto`.

The physical subtype of a package must be compatible with the vehicle's primary type and any physical subtype specified for that vehicle. Packages with specialty subtype of `hazardous` may be compatible with certain cities and incompatible with others. Hazardous packages cannot originate nor pass through cities unless that city is compatible with hazardous packages (specified via the predicate `pc-compatible`).

Packages have a location and fees to be collected. Hazardous packages require a permit and warning signs, and valuable packages require insurance.

7.2.1.2 Actions

This section describes the symbols that denote actions in UM Translog domain. Most symbol names are chosen to be self explanatory.

Prior to carrying a package to its destination, fees should be collected, a special permit should be obtained if the package is of type `hazardous`, the package should be insured if it is of type `valuable`, and all these should be cancelled once the package is delivered at its destination. These activities are denoted by the action symbols `obtain-permit(p)`, `collect-fees(p)`, `collect-insurance(p)`, and `deliver(p)`, where p is a variable symbol denoting a package.

There are a number of actions for loading and unloading packages from/to vehicles, depending on the type of vehicle and package. In some cases, special equipment such as cranes need to be used for that purpose.

Loading a *regular* package into a *regular* vehicle involves opening the door of the vehicle, putting the package in, and then closing the door, denoted by the actions `open-door(v)`, `load-package(p v l)`, `close-door(v)`. Unloading a regular package involves the same steps in reverse order, replacing load-package with `unload-package(p v l)`. `p` is a variable of type package, `v` is a variable of type vehicle, and `l` is a variable of type location. `l` is used to make sure the vehicle and the package are at the same location.

Packages of type *valuable* can be carried only by vehicles of type *armored*, and require the additional steps of posting a guard outside while loading, and posting a guard inside while in transportation and removing the guards afterwards are required. These are denoted by the primitive tasks `post-guard-outside(v)`, `post-guard-inside(v)`, and `remove-guard(v)`.

Packages of type *hazardous* can be carried only with proper warning signs on the vehicle, and the vehicle must be decontaminated afterwards. These actions are denoted by `affix-warning-signs (v)`, `remove-warning-signs (v)`, and `decontaminate-interior(v)`.

Loading/unloading a truck or traincar of type *flatbed* requires use of a crane denoted by c in the actions
`pick-up-package-ground(p c l)`, `put-down-package-ground(p c l)`,
`pick-up-package-vehicle(p c v l)`, and
`put-down-package-vehicle(p c v l)`.

Loading a truck or traincar of type *hopper* involves several actions:
`connect-chute(v)`, `fill-hopper(p v l)`, and `disconnect-chute(v)`. Unload is similar, simply replace `empty-hopper(p v l)` with `fill-hopper(p v l)`.

Loading/unloading vehicles of type *tanker* also involves several actions:
`connect-hose(v)`, `disconnect-hose(v p)`, `open-valve(v)`, `close-valve(v)`, `fill-tank(v p l)`, and `empty-tank(v p l)` in appropriate order.

Loading packages of type *livestock* involves the actions `lower-ramp(v)`, `fill-trough(v)`, `load-livestock(p v l)`, and `raise-ramp(v)`. Unloading involves the actions `lower-ramp(v)`, `unload-livestock(p v l)`, `raise-ramp(v)`, `do-clean-interior(v)`, and `unload-livestock(p v l)`.

Loading/unloading packages of type *cars* involves the actions `load-cars(p v l)`, and `unload-cars(p v l)`. As in the case of livestock, the ramp of the vehicle needs to be lowered prior to loading/unloading, and the it should be raised immediately afterwards.

Loading/unloading vehicles of type *airplane* requires a conveyor ramp (denoted by r) to be connected and the door of the vehicle to be open prior to the operation, and the ramp to be disconnected and the door to be closed afterwards. These activities are denoted by the actions `attach-conveyor-ramp(v r l)`, `detach-conveyor-ramp(v r l)`, `open-door(v)`,`close-door(v)`,
`load-package(p v l)`, and `load-package(p v l)`.

A vehicle v can be moved from its current location *ol* to another location *dl* if there is a route r of proper type between *ol* and *dl*, using the action `move-vehicle(v ol dl r)`. Vehicles of type *traincar* do not move by themselves but are pulled by vehicles of type *train* instead. Thus a traincar goes wherever the train it is attached to goes. The actions to attach/detach traincars to trains are `attach-train-car(t c l)`, and `detach-train-car(t c l)`.

In addition to those actions described above, UM Translog makes use of a dummy action called `do-nothing` which has no preconditions or effects.

7.2.1.3 Tasks

Figure 7.3 presents the organization of tasks and their subtasks, which are discussed below.

AT-PACKAGE(package destination) This task requires transporting the package to its destination.

TRANSPORT(package origin destination) Provided that the package is currently in the origin location, this task involves picking up the package, carrying it to its destination, and delivering it.

PICKUP(package) This task involves collecting fees, handling insurance and hazardous material permits. Insurance is required only for valuable packages, and permits are required

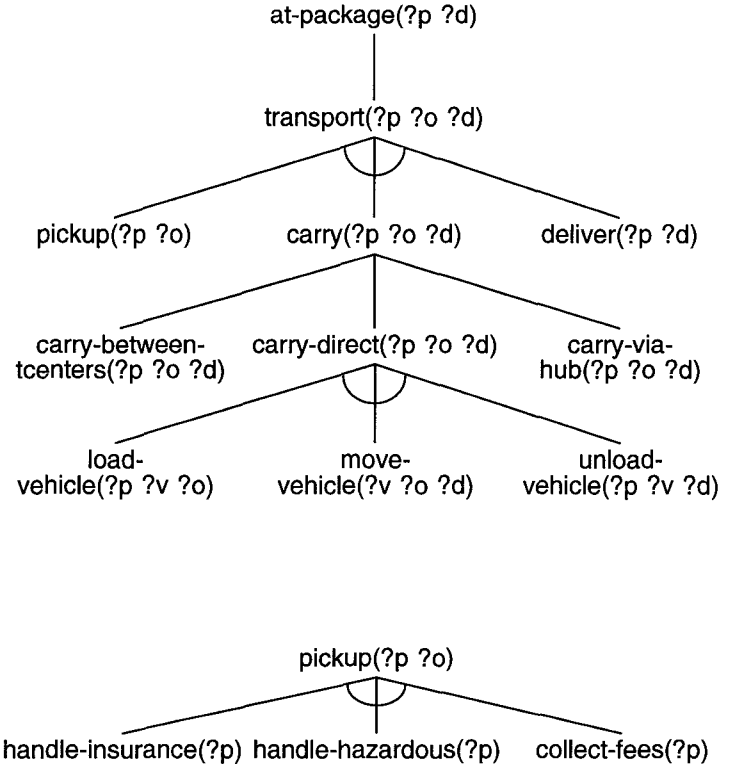


Figure 7.3: Top-level Task Hierarchy

only for hazardous packages.

CARRY(package origin destination) This is the task of actually moving the package from its origin to its destination. This involves choosing a suitable path (a sequence of routes from the origin to the destination), and moving the package along that path via a series of *carry-direct* tasks. The diagram in Figure 7.4 shows the possible paths to transport a package. The *origin* can be either *clocation1* (a customer location) or *tcenter1* (a transport center), and similarly the *destination* can be either *clocation2* or *tcenter2*. As seen in the diagram, a package can be carried directly if there is a direct route available, otherwise it has to go through one or two transportation centers and possibly a hb. Transport within a city is termed "local" transport. Transport via a direct route (i.e., not involving a hub) is termed "direct" transport. Transport via a hub is termed "indirect" transport.

CARRY-DIRECT(package origin destination) This task involves picking a route connecting the *origin* and the *destination*, and choosing a vehicle that is compatible both with the package, and the route. Only those vehicles that are at the *origin* or one step away from the *origin* can be dispatched. The task is accomplished by moving that vehicle to the origin, loading the package into the vehicle, moving the vehicle to the destination, and finally unloading the package.

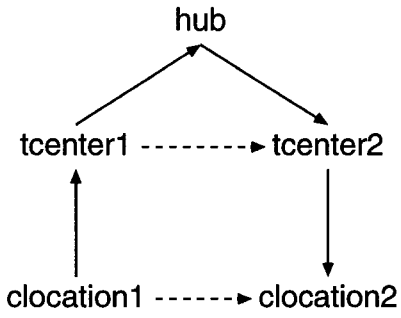


Figure 7.4: Transport Paths

AT-VEHICLE(vehicle destination) If the vehicle is of type truck, plane, or train, it is moved to the destination, provided there is a direct route available from the current location. The type of the route must be suitable for the type of vehicle. If the vehicle is a train-car, a train must be moved to the location of the train-car, the train-car must be attached to the train, the train must be moved to the destination, and then finally, the train-car must be detached. Naturally, when a vehicle moves, so does the packages it contains.

LOAD/UNLOAD(package vehicle location)

Loading and unloading involve issuing a sequence of actions to put the package into and out of the vehicle. The actions to be executed depend on the type of package and vehicle. In particular, valuable packages can be transported only with armored vehicles, and they require guards posted outside while loading/unloading and guards inside, while in transit. Similarly, vehicles carrying hazardous packages need warning signs, which are removed after the package is unloaded and the vehicle is decontaminated.

7.2.2 A Sample Problem in UM Translog Domain

The following illustrates the specification of an actual domain problem to UMCP. This sample problem involves delivering a regular, valuable package **pkg-1** to a customer location **city1-cl2**. Initially the package is at **city1-cl1**. Full description of the problem appears in the appendix.

The initial state for UM Translog problems are usually rather large, because in addition to the problem specification, it contains domain information regarding the compatibility of various package types, vehicle types, and route types. It also contains all the geographic information regarding the cities, routes, transportation centers and customer locations.

Here is how UMCP solves this problem:

USER(13):

USER(14): (search-for-plan in-tn)

```
N:
 AT-PACKAGE
 (PKG-1 CITY1-CL2)
```

Name: tn
Formula: T
Variables: NIL
Disjoint Variables: NIL
Postponed Orderings: NIL
Postponed State Constraints: NIL
Refinement Strategy: (EXPAND N)

This is the input task network. Since the constraint formula and all the auxiliary constraints are empty, UMCP chooses to expand that task.

```
N:
 do-nothing
 ()
```

Name: tn-1
Formula: (BEFORE (AT-PACKAGE PKG-1 CITY1-CL2) (FIRST N))
Variables: NIL
Disjoint Variables: NIL
Postponed Orderings: NIL
Postponed State Constraints: NIL
Refinement Strategy: (refine-constraint ((BEFORE (AT-PACKAGE PKG-1 CITY1-CL2) (FIRST N))))

One way of accomplishing AT-PACKAGE(PKG-1 CITY1-CL-2) is not to do anything, provided that it is already there, as seen in the constraint formula. This node gets pruned when this constraint is enforced, since the package is in a different location.

```
N1148:
 TRANSPORT
 (PKG-1 ?ORIGIN147 CITY1-CL2)
```

Name: tn-2
Formula: (AND (INITIALLY (AT-PACKAGE PKG-1 ?ORIGIN147)) (AFTER (AT-PACKAGE PKG-1 CITY1-CL2) (LAST N1148)))
Variables: ((?ORIGIN147 *all*))
Disjoint Variables: NIL
Postponed Orderings: NIL
Postponed State Constraints: NIL
Refinement Strategy: (refine-constraint ((INITIALLY (AT-PACKAGE PKG-1 ?ORIGIN147)) (AFTER (AT-PACKAGE PKG-1 CITY1-CL2) (LAST N1148))))

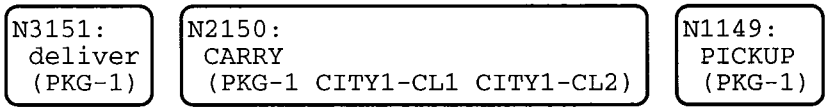
Thus UMCP tries to transport the package, which involves finding out its original location, denoted by the variable ?ORIGIN147. UMCP also needs to ensure the package will reach its destination in the end, as specified in the constraint formula.

```
N1148:
 TRANSPORT
 (PKG-1 CITY1-CL1 CITY1-CL2)
```

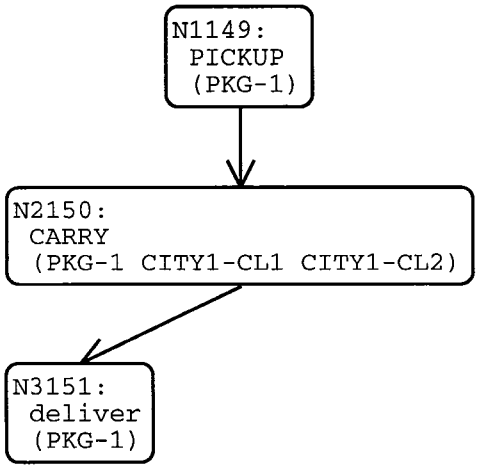
Name: tn-2-1

Formula: T
Variables: ((?ORIGIN147 CITY1-CL1))
Disjoint Variables: NIL
Postponed Orderings: NIL
Postponed State Constraints: ((AT-PACKAGE (AFTER (AT-PACKAGE PKG-1 CITY1-CL2) (LAST N1148))))
Refinement Strategy: (EXPAND N1148)

UMCP has determined that the origin of the package is CITY1-CL1, as can be seen from the "variables" list. It has promised to have the package at its destination in the end, as can be seen in the postponed state constraints. It suggests to expand the transport task next.



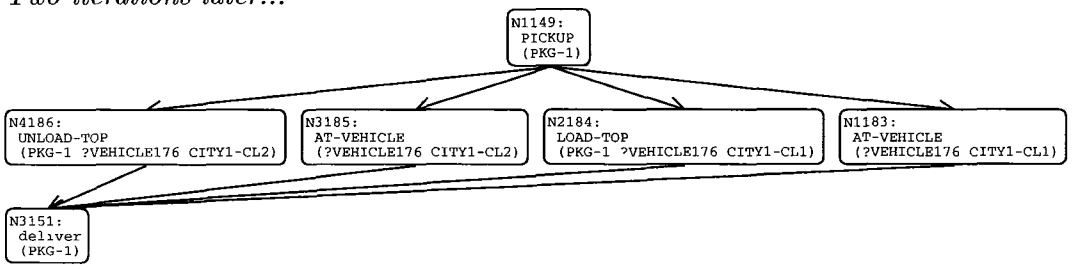
Name: tn-2-1-1
Formula: (AND (ORD (LAST N1149) (FIRST N2150)) (ORD (LAST N2150) (FIRST N3151)))
Variables: ((?ORIGIN147 CITY1-CL1))
Disjoint Variables: NIL
Postponed Orderings: NIL
Postponed State Constraints: ((AT-PACKAGE (AFTER (AT-PACKAGE PKG-1 CITY1-CL2) (LAST N3151 N2150 N1149))))
Refinement Strategy: (refine-constraint ((ORD (LAST N2150) (FIRST N3151)) (ORD (LAST N1149) (FIRST N2150))))



Name: tn-2-1-1-1
Formula: T
Variables: ((?ORIGIN147 CITY1-CL1))
Disjoint Variables: NIL
Postponed Orderings: NIL
Postponed State Constraints: ((AT-PACKAGE (BEFORE (AT-PACKAGE PKG-1 CITY1-CL2) (LAST N3151))))
Refinement Strategy: (EXPAND N2150)

Processing the constraints selected at the previous iteration has caused UMCP to order the tasks.

Two iterations later...



Name: tn-2-1-1-1-1-1

Formula:

(AND (ORD (LAST N1183) (FIRST N2184)) (ORD (LAST N2184) (FIRST N3185))
(ORD (LAST N3185) (FIRST N4186))
(BEFORE (AT-PACKAGE PKG-1 CITY1-CL1) (FIRST N2184))
(INITIALLY (TYPE PKG-1 ?PTYPE177))
(BETWEEN (AT-VEHICLE ?VEHICLE176 CITY1-CL1) (LAST N1183) (FIRST N2184))
(BETWEEN (AT-PACKAGE PKG-1 ?VEHICLE176) (LAST N2184) (FIRST N4186))
(INITIALLY (AVAILABLE ?VEHICLE176))
(INITIALLY (TYPE ?VEHICLE176 ?VTYPE178))
(INITIALLY (PV-COMPATIBLE ?PTYPE177 ?VTYPE178))
(OR (AND (INITIALLY (TYPE ?VEHICLE176 TRUCK))
(INITIALLY (IN-CITY CITY1-CL1 ?OCITY179))
(INITIALLY (IN-CITY CITY1-CL2 ?OCITY179)))
(AND (INITIALLY (TYPE ?VEHICLE176 TRUCK))
(INITIALLY (IN-CITY CITY1-CL1 ?OCITY179))
(INITIALLY (IN-CITY CITY1-CL2 ?DCITY180))
(INITIALLY (CONNECTS ?ROUTE182 ROAD-ROUTE ?OCITY179 ?DCITY180))
(INITIALLY (AVAILABLE ?ROUTE182)))
(AND (INITIALLY (¬TYPE ?VEHICLE176 TRUCK))
(INITIALLY (CONNECTS ?ROUTE182 ?RTYPE181 CITY1-CL1 CITY1-CL2))
(INITIALLY (RV-COMPATIBLE ?RTYPE181 ?VHTYPE))
(INITIALLY (TYPE ?VEHICLE176 ?VHTYPE))
(INITIALLY (AVAILABLE ?ROUTE182)))))

Variables: ((?VEHICLE176 *all*) (?PTYPE177 *all*) (?VTYPE178 *all*) (?OCITY179 *all*) (?DCITY180 *all*) (?RTYPE181 *all*) (?ROUTE182 *all*) (?ORIGIN147 CITY1-CL1))

Disjoint Variables: NIL

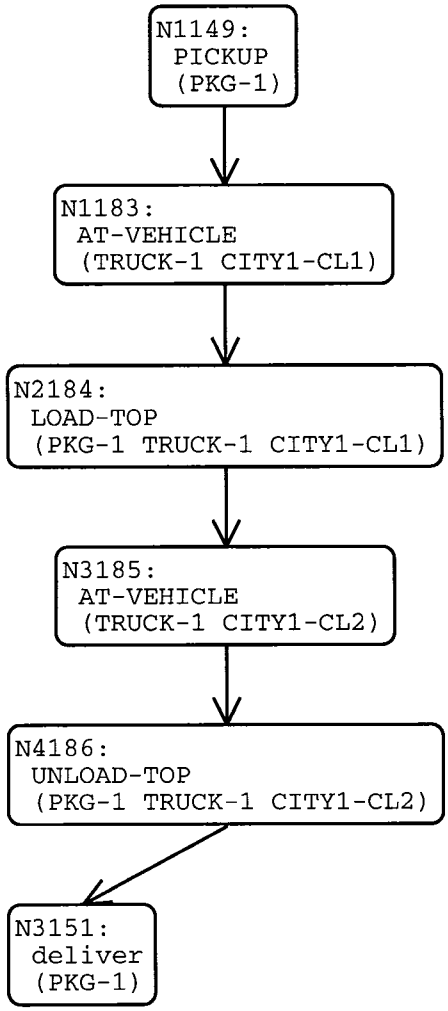
Postponed Orderings: NIL

Postponed State Constraints: ((AT-PACKAGE (BEFORE (AT-PACKAGE PKG-1 CITY1-CL2) (LAST N3151))))

Refinement Strategy: (refine-constraint
((INITIALLY (PV-COMPATIBLE ?PTYPE177 ?VTYPE178))
(INITIALLY (TYPE ?VEHICLE176 ?VTYPE178))
(INITIALLY (AVAILABLE ?VEHICLE176))
(INITIALLY (TYPE PKG-1 ?PTYPE177))

(ORD (LAST N3185) (FIRST N4186))
(ORD (LAST N2184) (FIRST N3185))
(ORD (LAST N1183) (FIRST N2184))
(BEFORE (AT-PACKAGE PKG-1 CITY1-CL1) (FIRST N2184))
(BETWEEN (AT-PACKAGE PKG-1 ?VEHICLE176) (LAST N2184) (FIRST N4186))
(BETWEEN (AT-VEHICLE ?VEHICLE176 CITY1-CL1) (LAST N1183) (FIRST N2184))))

At this stage the task networks start to get quite big. UMCP picks all the unit clauses in the constraint formula for refinement.



Name: tn-2-1-1-1-1-1-1

Formula: (OR (AND (INITIALLY (IN-CITY CITY1-CL1 ?OCITY179))
(INITIALLY (IN-CITY CITY1-CL2 ?OCITY179)))
(AND (INITIALLY (IN-CITY CITY1-CL1 ?OCITY179))
(INITIALLY (IN-CITY CITY1-CL2 ?DCITY180))
(INITIALLY (CONNECTS ?ROUTE182 ROAD-ROUTE ?OCITY179 ?DCITY180))
(INITIALLY (AVAILABLE ?ROUTE182))))

Variables: ((?ORIGIN147 CITY1-CL1) (?ROUTE182 *all*) (?RTYPE181 *all*) (?DCITY180 *all*) (?OCITY179 *all*) (?VTYPE178 ARMORED) (?PTYPE177 VALUABLE) (?VEHICLE176

TRUCK-1))

Disjoint Variables: NIL

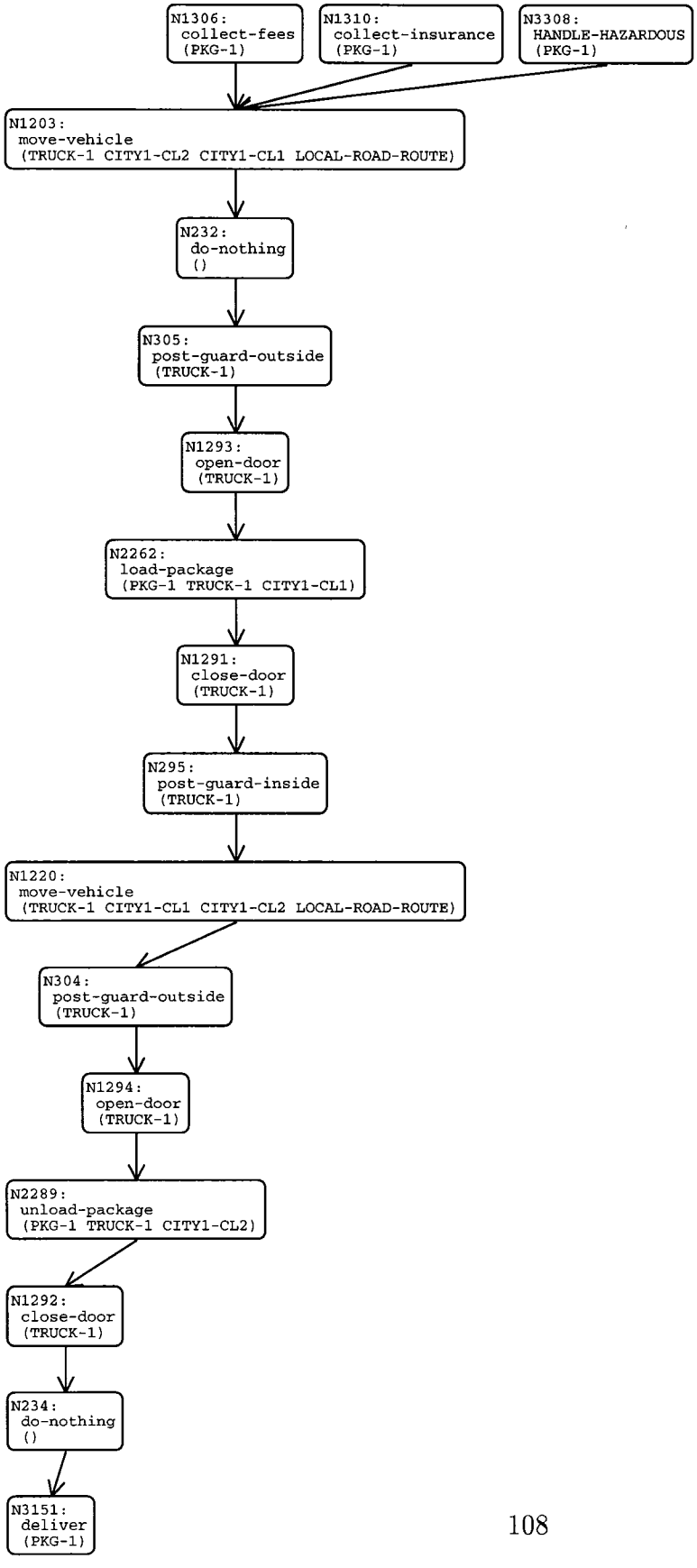
Postponed Orderings: NIL

Postponed State Constraints: ((AT-VEHICLE (AFTER (AT-VEHICLE ?VEHICLE176 CITY1-CL1) (LAST N1183)))
(AT-PACKAGE (BEFORE (AT-PACKAGE PKG-1 CITY1-CL2) (LAST N3151))
(AFTER (AT-PACKAGE PKG-1 ?VEHICLE176) (LAST N2184))))

Refinement Strategy: (refine-constraint ((INITIALLY (IN-CITY CITY1-CL1 ?OCITY179))) ((NOT (INITIALLY (IN-CITY CITY1-CL1 ?OCITY179)))))

UMCP has processed the constraints selected in the previous iteration, and determined the values of some of the variables. Since the constraint formula contains a disjunction, UMCP has picked an atomic constraint and its negation to work on further in separate branches.

Many iterations later...


N1306:
collect-fees
(PKG-1)
N1310:
collect-insurance
(PKG-1)
N3308:
HANDLE-HAZARDOUS
(PKG-1)
N1203:
move-vehicle
(TRUCK-1 CITY1-CL2 CITY1-CL1 LOCAL-ROAD-ROUTE)
N232:
do-nothing
()
N305:
post-guard-outside
(TRUCK-1)
N1293:
open-door
(TRUCK-1)
N2262:
load-package
(PKG-1 TRUCK-1 CITY1-CL1)
N1291:
close-door
(TRUCK-1)
N295:
post-guard-inside
(TRUCK-1)
N1220:
move-vehicle
(TRUCK-1 CITY1-CL1 CITY1-CL2 LOCAL-ROAD-ROUTE)
N304:
post-guard-outside
(TRUCK-1)
N1294:
open-door
(TRUCK-1)
N2289:
unload-package
(PKG-1 TRUCK-1 CITY1-CL2)
N1292:
close-door
(TRUCK-1)
N234:
do-nothing
()
N3151:
deliver
(PKG-1)

Name: tn-2-1-1-1-1-1-1-1-2-1-2-2-1-2-1-1-1-1-2-1-2-1-2-1-2-1-1-1-1-1-2-2-2-2-2-2-2-1-1

Formula: (INITIALLY (TYPE PKG-1 VALUABLE))

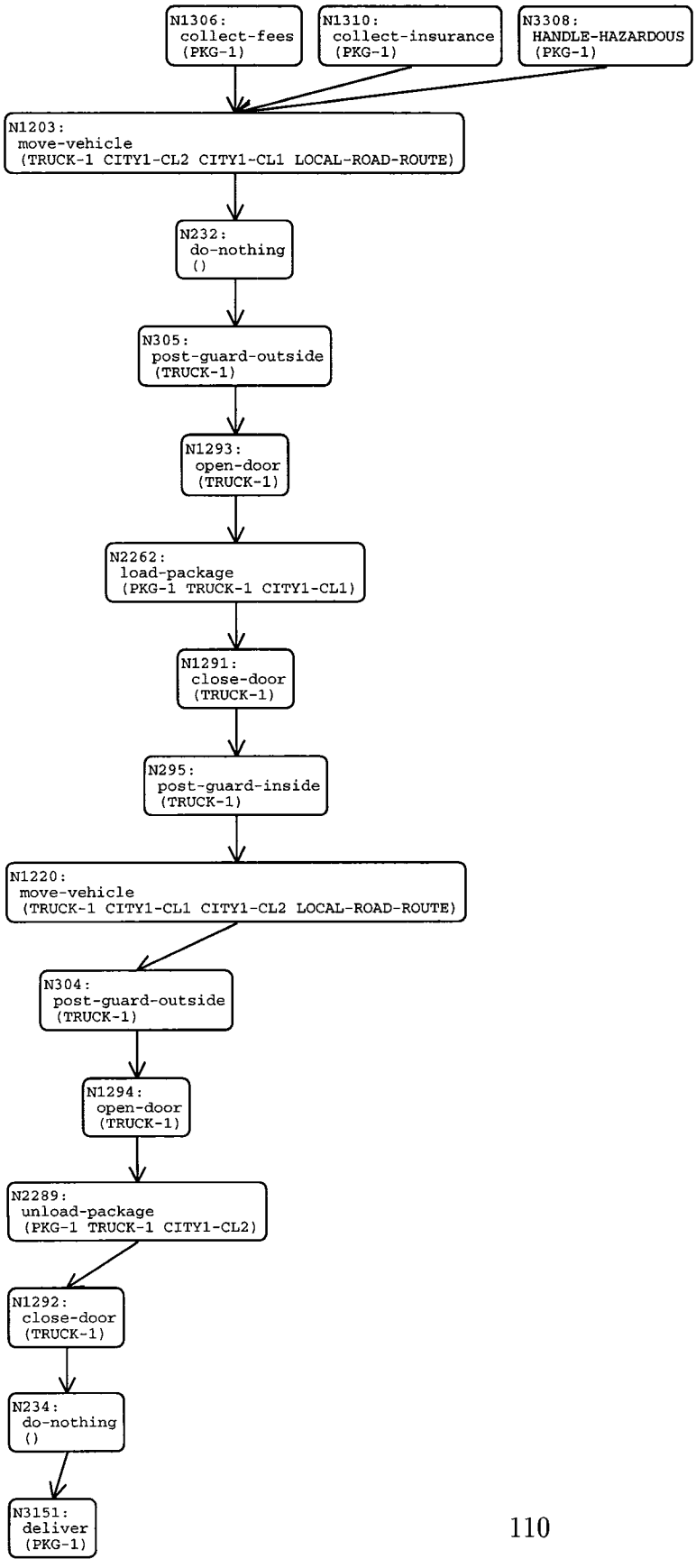
Variables: ((?ORIGIN147 CITY1-CL1) (?ROUTE182 *all*) (?RTYPE181 *all*) (?DCITY180 *all*) (?OCITY179 CITY1) (?VTYPE178 ARMORED) (?PTYPE177 VALUABLE) (?VEHICLE176 TRUCK-1) (?DCITY202 *all*) (?OCITY201 CITY1) (?ORIGIN200 CITY1-CL2) (?R199 LOCAL-ROAD-ROUTE) (?DCITY219 *all*) (?OCITY218 CITY1) (?ORIGIN217 CITY1-CL1) (?R216 LOCAL-ROAD-ROUTE))

Disjoint Variables: NIL

Postponed Orderings: NIL

Postponed State Constraints:
((AT-VEHICLE (AFTER (AT-VEHICLE TRUCK-1 CITY1-CL1) (LAST N1293))) (DOOR-OPEN (AFTER (DOOR-OPEN TRUCK-1) (LAST N1294)) (AFTER (DOOR-OPEN TRUCK-1) (LAST N1293))) (AT-PACKAGE (AFTER (AT-PACKAGE PKG-1 CITY1-CL1) (LAST N1293)) (AFTER (AT-PACKAGE PKG-1 ?VEHICLE176) (LAST N295))))

Refinement Strategy: (refine-constraint ((INITIALLY (TYPE PKG-1 VALUABLE))))


N1306:
collect-fees
(PKG-1)
N1310:
collect-insurance
(PKG-1)
N3308:
HANDLE-HAZARDOUS
(PKG-1)
N1203:
move-vehicle
(TRUCK-1 CITY1-CL2 CITY1-CL1 LOCAL-ROAD-ROUTE)
N232:
do-nothing
()
N305:
post-guard-outside
(TRUCK-1)
N1293:
open-door
(TRUCK-1)
N2262:
load-package
(PKG-1 TRUCK-1 CITY1-CL1)
N1291:
close-door
(TRUCK-1)
N295:
post-guard-inside
(TRUCK-1)
N1220:
move-vehicle
(TRUCK-1 CITY1-CL1 CITY1-CL2 LOCAL-ROAD-ROUTE)
N304:
post-guard-outside
(TRUCK-1)
N1294:
open-door
(TRUCK-1)
N2289:
unload-package
(PKG-1 TRUCK-1 CITY1-CL2)
N1292:
close-door
(TRUCK-1)
N234:
do-nothing
()
N3151:
deliver
(PKG-1)

Name: tn-2-1-1-1-1-1-1-1-2-1-2-2-1-2-1-1-1-1-2-1-2-1-2-1-2-1-1-1-1-1-2-2-2-2-2-2-2-1-1-1

Formula: T

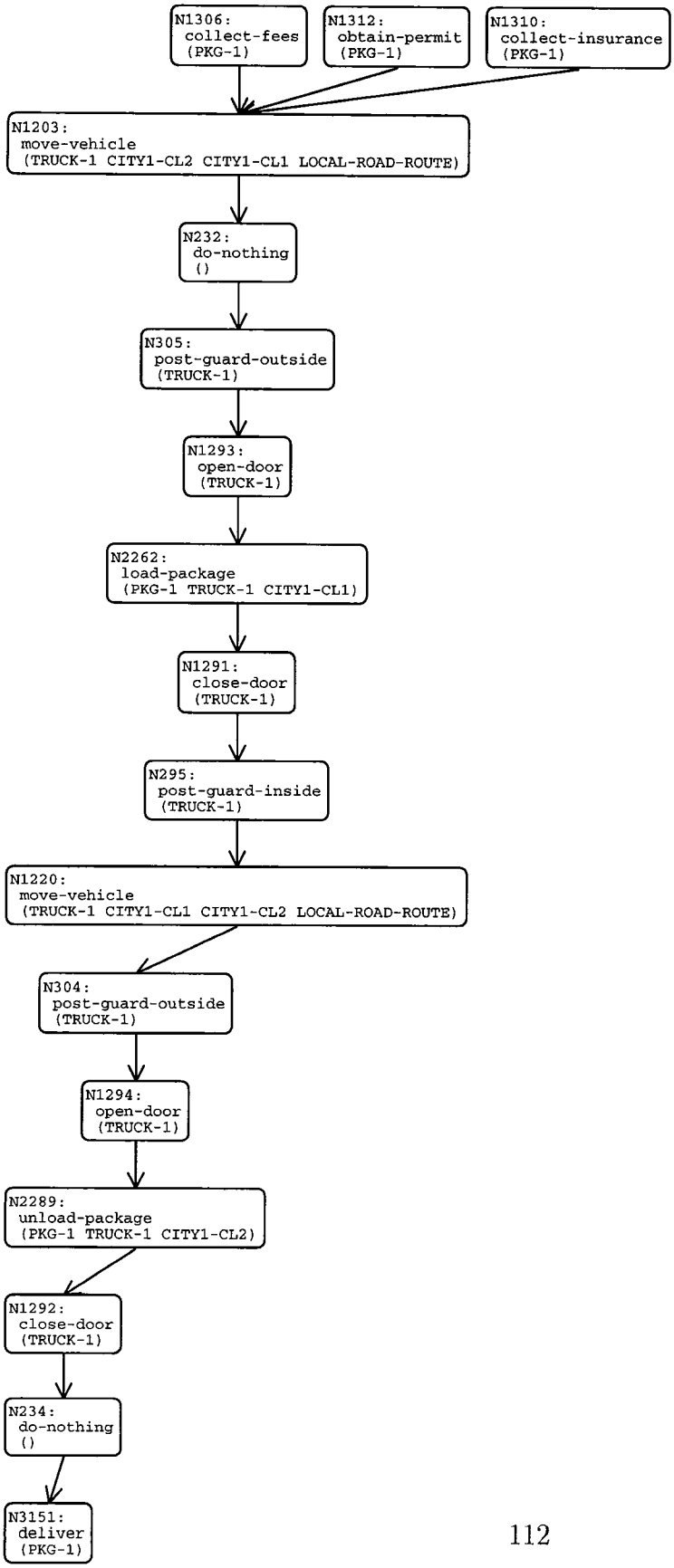
Variables: ((?ORIGIN147 CITY1-CL1) (?ROUTE182 *all*) (?RTYPE181 *all*) (?DCITY180 *all*) (?OCITY179 CITY1) (?VTYPE178 ARMORED) (?PTYPE177 VALUABLE) (?VEHICLE176 TRUCK-1) (?DCITY202 *all*) (?OCITY201 CITY1) (?ORIGIN200 CITY1-CL2) (?R199 LOCAL-ROAD-ROUTE) (?DCITY219 *all*) (?OCITY218 CITY1) (?ORIGIN217 CITY1-CL1) (?R216 LOCAL-ROAD-ROUTE))

Disjoint Variables: NIL

Postponed Orderings: NIL

Postponed State Constraints: NIL

Refinement Strategy: (EXPAND N3308)


N1306:
collect-fees
(PKG-1)
N1312:
obtain-permit
(PKG-1)
N1310:
collect-insurance
(PKG-1)
N1203:
move-vehicle
(TRUCK-1 CITY1-CL2 CITY1-CL1 LOCAL-ROAD-ROUTE)
N232:
do-nothing
()
N305:
post-guard-outside
(TRUCK-1)
N1293:
open-door
(TRUCK-1)
N2262:
load-package
(PKG-1 TRUCK-1 CITY1-CL1)
N1291:
close-door
(TRUCK-1)
N295:
post-guard-inside
(TRUCK-1)
N1220:
move-vehicle
(TRUCK-1 CITY1-CL1 CITY1-CL2 LOCAL-ROAD-ROUTE)
N304:
post-guard-outside
(TRUCK-1)
N1294:
open-door
(TRUCK-1)
N2289:
unload-package
(PKG-1 TRUCK-1 CITY1-CL2)
N1292:
close-door
(TRUCK-1)
N234:
do-nothing
()
N3151:
deliver
(PKG-1)

Name: tn-2-1-1-1-1-1-1-1-2-1-2-2-1-2-1-1-1-1-2-1-2-1-2-1-2-1-1-1-1-1-2-2-2-2-2-2-2-1-1-1-1

Formula: (INITIALLY (TYPE PKG-1 HAZARDOUS) N1312)

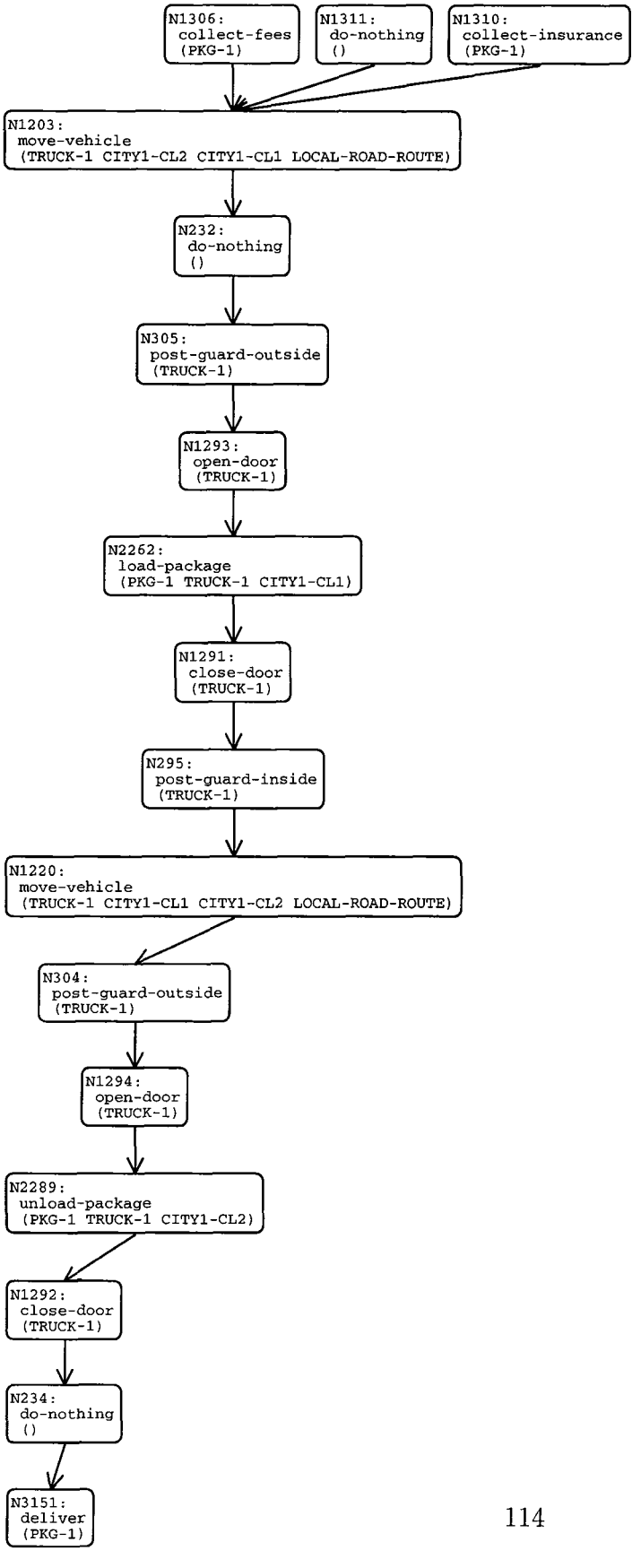
Variables: ((?ORIGIN147 CITY1-CL1) (?ROUTE182 *all*) (?RTYPE181 *all*) (?DCITY180 *all*) (?OCITY179 CITY1) (?VTYPE178 ARMORED) (?PTYPE177 VALUABLE) (?VEHICLE176 TRUCK-1) (?DCITY202 *all*) (?OCITY201 CITY1) (?ORIGIN200 CITY1-CL2) (?R199 LOCAL-ROAD-ROUTE) (?DCITY219 *all*) (?OCITY218 CITY1) (?ORIGIN217 CITY1-CL1) (?R216 LOCAL-ROAD-ROUTE))

Disjoint Variables: NIL

Postponed Orderings: NIL

Postponed State Constraints: NIL

Refinement Strategy: (refine-constraint ((INITIALLY (TYPE PKG-1 HAZARDOUS) N1312)))


N1306:
collect-fees
(PKG-1)
N1311:
do-nothing
()
N1310:
collect-insurance
(PKG-1)
N1203:
move-vehicle
(TRUCK-1 CITY1-CL2 CITY1-CL1 LOCAL-ROAD-ROUTE)
N232:
do-nothing
()
N305:
post-guard-outside
(TRUCK-1)
N1293:
open-door
(TRUCK-1)
N2262:
load-package
(PKG-1 TRUCK-1 CITY1-CL1)
N1291:
close-door
(TRUCK-1)
N295:
post-guard-inside
(TRUCK-1)
N1220:
move-vehicle
(TRUCK-1 CITY1-CL1 CITY1-CL2 LOCAL-ROAD-ROUTE)
N304:
post-guard-outside
(TRUCK-1)
N1294:
open-door
(TRUCK-1)
N2289:
unload-package
(PKG-1 TRUCK-1 CITY1-CL2)
N1292:
close-door
(TRUCK-1)
N234:
do-nothing
()
N3151:
deliver
(PKG-1)

Name: tn-2-1-1-1-1-1-1-1-2-1-2-2-1-2-1-1-1-1-2-1-2-1-2-1-2-1-1-1-1-1-2-2-2-2-2-2-2-1-1-1-2

Formula: (INITIALLY (¬TYPE PKG-1 HAZARDOUS))

Variables: ((?ORIGIN147 CITY1-CL1) (?ROUTE182 *all*) (?RTYPE181 *all*) (?DCITY180 *all*) (?OCITY179 CITY1) (?VTYPE178 ARMORED) (?PTYPE177 VALUABLE) (?VEHICLE176 TRUCK-1) (?DCITY202 *all*) (?OCITY201 CITY1) (?ORIGIN200 CITY1-CL2) (?R199 LOCAL-ROAD-ROUTE) (?DCITY219 *all*) (?OCITY218 CITY1 *all*) (?ORIGIN217 CITY1-CL1) (?R216 LOCAL-ROAD-ROUTE))

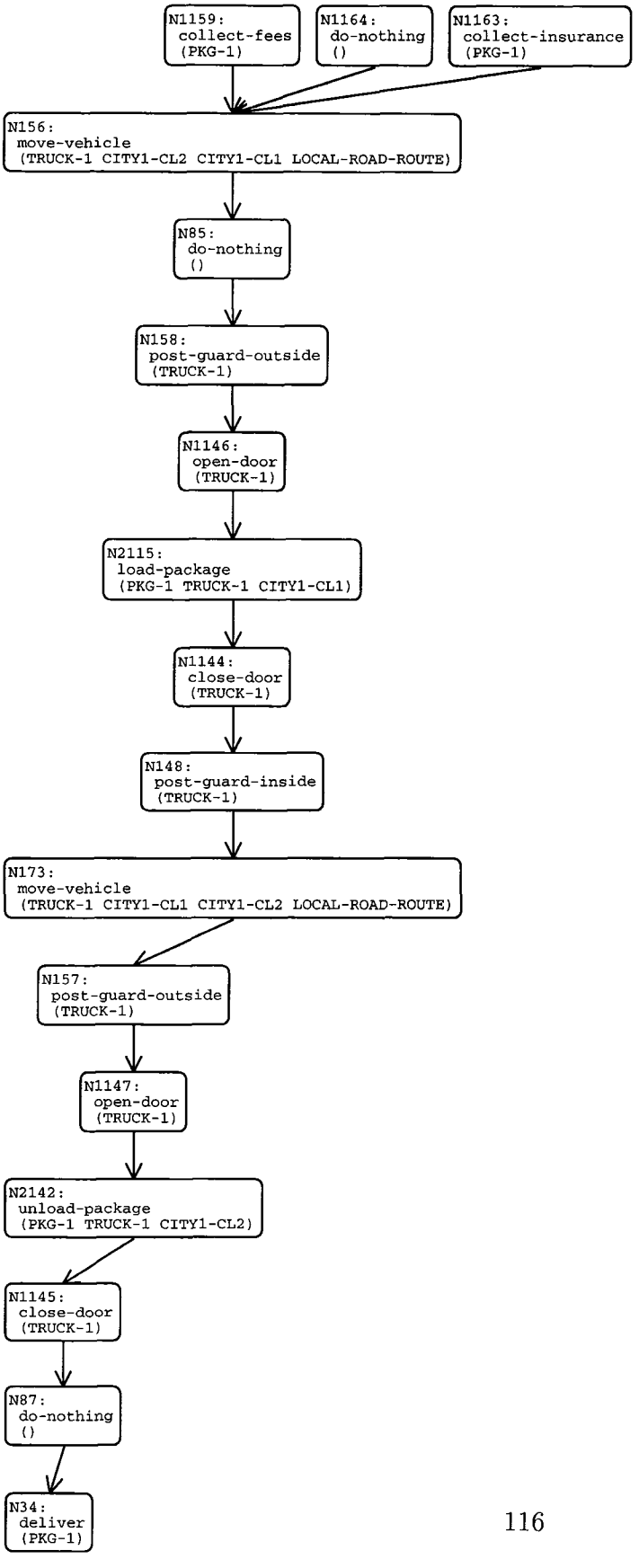
Disjoint Variables: NIL

Postponed Orderings: NIL

Postponed State Constraints: NIL

Refinement Strategy: (refine-constraint ((INITIALLY (¬TYPE PKG-1 HAZARDOUS))))

Here is one solution


N1159:
collect-fees
(PKG-1)
N1164:
do-nothing
()
N1163:
collect-insurance
(PKG-1)
N156:
move-vehicle
(TRUCK-1 CITY1-CL2 CITY1-CL1 LOCAL-ROAD-ROUTE)
N85:
do-nothing
()
N158:
post-guard-outside
(TRUCK-1)
N1146:
open-door
(TRUCK-1)
N2115:
load-package
(PKG-1 TRUCK-1 CITY1-CL1)
N1144:
close-door
(TRUCK-1)
N148:
post-guard-inside
(TRUCK-1)
N173:
move-vehicle
(TRUCK-1 CITY1-CL1 CITY1-CL2 LOCAL-ROAD-ROUTE)
N157:
post-guard-outside
(TRUCK-1)
N1147:
open-door
(TRUCK-1)
N2142:
unload-package
(PKG-1 TRUCK-1 CITY1-CL2)
N1145:
close-door
(TRUCK-1)
N87:
do-nothing
()
N34:
deliver
(PKG-1)

Name: tn-2-1-1-1-1-1-1-1-2-1-2-2-1-2-1-1-1-1-2-1-2-1-2-1-2-1-1-1-1-1-2-2-2-2-2-2-2-1-1-1-2-1

Formula: T

Variables: ((?ORIGIN0 CITY1-CL1) (?ROUTE35 ALL) (?RTYPE34 ALL) (?DCITY33 ALL) (?OCITY32 CITY1) (?VTYPE31 ARMORED) (?PTYPE30 VALUABLE) (?VEHICLE29 TRUCK-1) (?DCITY55 ALL) (?OCITY54 CITY1) (?ORIGIN53 CITY1-CL2) (?R52 LOCAL-ROAD-ROUTE) (?DCITY72 ALL) (?OCITY71 CITY1) (?ORIGIN70 CITY1-CL1) (?R69 LOCAL-ROAD-ROUTE))

Disjoint Variables: NIL

Postponed Orderings: NIL

Postponed State Constraints: NIL

Search for more (y/n)? n

USER(15):

In solving this problem, UMCP has generated 85 nodes, and expanded 63 nodes. The distance of the solution to the input node is 42 refinements. As can be seen from the name of the solution task network, UMCP deviated from the solution path 17 times, but the constraint refinement strategies detected dead-ends in very few steps, with an avarage of $(63 - 42)/17 = 1.24$ steps.

7.3 Case Study 2: CNF Domain

CNF domain is an artificial domain, which can be easily encoded in the HTN language, but impossible to represent in the state-based planning framework.

This domain is designed so that the rules for any two contextfree grammars can be represented in the initial state, and the plans correspond to strings that are common to the langugages generated by the corresponding grammars.

This is the domain that was used in the proof of Theorem 5 to demonstrate that there exists planning domains that can be encoded in the HTN language for which planning is semidecidable. A detailed explanation of this domain is presented in Appendix A, in the proof of Theorem 5.

7.3.1 Domain Specification for the CNF Domain

(clear-domain)

(constants a1 a2 b1 b2 S1 S2 Q1 Q2 dummy)

Si and Qi are the nonterminal grammar symbols ai and bi are the terminal grammar symbols. dummy corresponds to the empty string.

(variables v v1 v2)

(predicates Pa Pb R turn)

Proposition Pa and Pb are used to represent the terminal symbol contributed by the first grammar, to enforce the second grammar to contribute the same symbol. Proposition turn is used to make sure each grammar contributes a terminal symbol alternatingly. Predicate R is used to encode grammar rules.

(primitive-tasks Fa1 Fa2 Fb1 Fb2)

(compound-tasks G)

G has arity 1, its argument is a grammar symbol. If the argument ,is a terminal symbol, then it expands to the corresponding primitive task. If it is a nonterminal symbol, then it can expand to any sequence of primitive tasks, which correspond to the strings that can be derived from that nonterminal symbol.

```
(operator Fa1()
  :pre ((~turn))
  :post ((Pa)(turn))
)\\
(operator Fa2()\\
  :pre ((Pa))\\
  :post ((~Pa)(~turn))
)\\

(operator Fb1()
  :pre ((~turn))
  :post ((Pb)(turn))
)
(operator Fb2()
  :pre ((Pb))
  :post((~Pb)(~turn))
)

(declare-method G(v)
:expansion( (n Fa1 ) )
:formula (veq v a1)
)

(declare-method G(v)
:expansion( (n Fa2 ) )
:formula (veq v a2)
)

(declare-method G(v)
:expansion( (n Fb1 ) )
:formula (veq v b1)
)
(declare-method G(v)
:expansion( (n Fb2 ) )
:formula (veq v b2)
)

(declare-method G(v)
:expansion( (n do-nothing ) )
```

```
:formula (veq v dummy)
)

(declare-method G(v)
:expansion( (n1 G v1)(n2 G v2))
:formula (and
  (ord n1 n2)
  (initially (R v v1 v2))))
```

7.3.2 A Sample Problem

Let us define two context-free languages, and ask UMCP to find a string that is common to both.

The first language is $\{a^n b^n | n \geq 0\}$. The corresponding grammar is

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow aQ \\ Q &\rightarrow Sb \end{aligned}$$

The second language is $\{(ab)^n | n \geq 1\}$. The corresponding grammar is

$$\begin{aligned} S &\rightarrow ab \\ S &\rightarrow aQ \\ Q &\rightarrow bS \end{aligned}$$

Below is the encoding of these two grammars in the initial state of the sample problem.

```
(initially-true
;grammar 1
 (R S1 dummy dummy)
 (R S1 a1    Q1   )
 (R Q1 S1    b1   )
;grammar 2
 (R S2 a2    b2   )
 (R S2 a2    Q2   )
 (R Q2 b2    S2   )
)


(setq g-tn (create-tn
     (after (~turn) (last n1 step1))
   (n1 G S1)(step1 G S2))
      )
```

The input task network contains the tasks corresponding to the start symbols of the two context-free languages, and the constraint in the constraint formula ensures that the string generated by S1 and S2 has the same length.

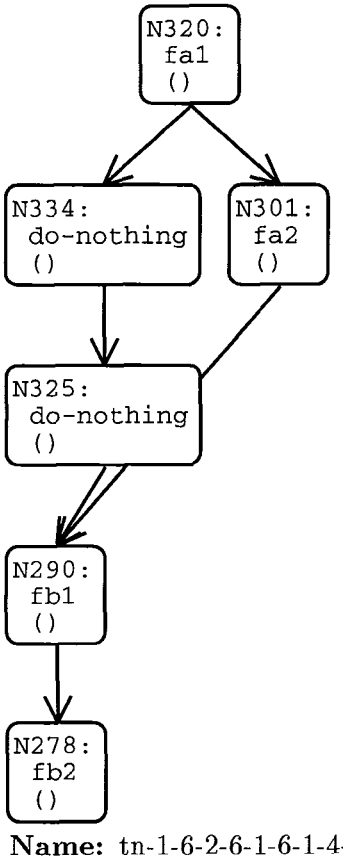
7.3.3 Solving the Sample Problem with UMCP

The CNF domain, in spite of its simplicity, is much harder to plan than the UM Translog domain. UMCP finds the solution string *ab* only after 113 node expansions.

USER(13):
USER(14): (search-for-plan g-tn)

This is the initial problem:

Name: tn
Formula: (AFTER (TURN) (LAST N1 STEP1))
Variables: NIL
Disjoint Variables: NIL
Postponed Orderings: NIL
Postponed State Constraints: NIL
Refinement Strategy: (9 ((AFTER (TURN) (LAST N1 STEP1))))
Here is the solution:



Name: tn-1-6-2-6-1-6-1-4-1-3-1-2-1-6-1-1-1-5-1-5-1
Formula: T
Variables: ((V2303 DUMMY) (V1304 DUMMY) (V1167 A1) (V2166 Q1) (V1254 A2) (V2253 B2) (V1264 S1) (V2263 B1))

Disjoint Variables: NIL
Postponed Orderings: NIL
Postponed State Constraints: NIL

Chapter 8

Conclusion

8.1 Research Contributions

Planning has a wide range of applications in areas such as robotics, manufacturing, space missions, and military force deployment. The size and complexity of these applications are rapidly growing beyond the capabilities of the people responsible for planning. There is great economic pressure to automate this process in order to reduce the cost of planning effort, and to improve the quality of produced plans.

AI planning research has not yet met these challenges. Part of the reason, in my opinion, has been the gap between theoretical work and application-oriented work on planning. Lack of interaction between planning theory and practice has impeded progress in both.

A significant portion of the application-oriented work in planning has been based on HTN techniques. The fundamental contribution of this dissertation has been to provide a theoretical framework for HTN planning. The HTN framework consists of a formal HTN language and the associated syntax and semantics. This framework facilitates learning about HTN planning, writing correct HTN problem specifications, developing correct HTN planning algorithms, and also enables further analytical studies on HTN planning. Each of these points will be explained in subsequent paragraphs.

The HTN framework provides a precise, consistent and coherent meaning to each HTN construct. The HTN paradigm comes with many powerful concepts and constructs such as compound tasks to model complex planning jobs, methods as a declarative construct to define tasks, filter conditions to weed out undesirable solutions and prune the search space. Previously, the meaning of these constructs were embedded in the implementation details of HTN planning systems, and their meanings varied depending on the planning system. For example, task expansions had the issue of what to do with high-level effects and constraints associated with the expanded task. Planning systems usually made an arbitrary decision to designate one of the tasks in the expansion to carry over the constraints from the expanded task. The HTN framework addresses this issue by introducing constructs that make it possible to refer to the beginning and end of a single task or a set of tasks. Thus the constraints on an expanded task could be passed over to the set of tasks in the expansion.

HTN constructs are powerful, but also complex. It is rather difficult to learn those constructs by examining the source code for implemented planning systems, or even their user manuals. The HTN framework provides a mathematical model, which can be taught independent of the implementation features of complex planning systems. Although the HTN constructs are complex, the concepts are quite intuitive. The preciseness, consistency and coherency of the framework makes learning easier. The UMCP planning system complements the learning process by showing the HTN planning process in action. It (optionally) shows what the planner does at every step, so that the HTN constructs can be better understood. The HTN framework has already made its way into an acclaimed AI text book, thanks to Russell and Norvig [57].

Writing correct domain specifications is a very challenging job, particularly for large applications. It can be even more difficult, if there are doubts about how the domain specifications are going to be interpreted by the planner. When the produced solutions are incorrect, it may take considerable effort to find out whether there is a mistake in the domain specifications, or a bug in the planner code. For example, developing specifications for the UM Translog Domain [4] involved several months of debugging both the NONLIN planning system, and the domain specification. Developing specifications for the same domain in the HTN language took little effort, even considering the experience from the same job on NONLIN. The HTN framework helps writing domain specifications by providing a precise model, where the entities in an application domain can be cast into.

The operational semantics in the HTN framework facilitates developing correct algorithms for planning systems, by providing a specification for the set of solutions to HTN problems. It also gives a criterion for evaluating HTN algorithms for soundness and completeness. The development of the UMCP planning system has been guided by the HTN framework, and thus it could be made sound and complete.

Analytical tools can be a big asset in studying and improving HTN planning systems. The HTN framework has made it possible to use such tools to get a deeper understanding of HTN planning. The complexity and expressivity analyses presented in this dissertation serve to this purpose.

This dissertation contains a complexity analysis of HTN planning, which brings a deep understanding of the factors that contribute to the computational cost of solving HTN problems. This analysis reveals that handling interactions among subtasks, and particularly those interactions resulting from state constraints are the most complex aspect of solving HTN problems. Limiting those interactions – for instance by restricting all task networks to be totally ordered – significantly reduces the complexity. The complexity of STRIPS-style planning is mostly influenced by the size and structure of the state space. The HTN complexity analysis shows that the structure of compound tasks is a much more important factor compared to the state space, in the complexity of solving HTN problems. Particularly those tasks which can be accomplished in infinitely many ways (i.e., recursive tasks, whose expansions may contain themselves) are difficult to deal with.

The definitions developed to compare the expressivity of planning languages have been used to investigate the similarities and differences between state-based planning and HTN

planning. This study enhances our understanding of both paradigms, and facilitates transfer of results from one paradigm to the other. It reveals that the relationship between state-based planning and HTN planning is akin to the relationship between regular languages and context-free languages. The transformations from the STRIPS language to the HTN language have been used to prove some of the complexity results on HTN planning.

The definition of solutions to HTN planning problems, and the insights gained from the analyses results have been used to develop a provably correct HTN algorithm, which have been implemented in the UMCP planning system. The UMCP planning system serves as a testbed for conducting experiments to investigate new ideas on HTN planning. It can also be a valuable education tool for teaching HTN planning.

8.2 Future Research Directions

The current state of the art in planning research has not yet reached a level to accommodate the demands of the planning applications. Developing fast, reliable planning systems that work well in planning applications is still a great challenge for planning researchers. HTN planning paradigm is a significant improvement over state-based planning. Nonetheless, it needs to be extended and improved in several ways to meet the challenges from planning applications, as outlined in the following sections.

8.2.1 Improving Performance of HTN Planning

HTN planning constructs can represent rather complex planning problems. Solving those problems in a reasonable amount of time is very difficult. The UMCP planning system meets part of this challenge by providing ways domain-specific information can be utilized to improve its performance. Two of these ways are user-specific critics and high-level search heuristics. There are several other ways that need to be explored.

The choice of commitment strategy is very critical in determining the structure of the search space and thus the overall performance of planning systems. A planner making decisions at the right order can be tremendously more efficient than a planner making these decisions in an arbitrary order. There is already a significant body of work in state-based planning framework on abstraction hierarchies. This work can be extended to the HTN framework, and it can provide good ways of deciding which task to expand first. Refinement strategies in HTN planning are numerous: a planner can either pick a task to expand, a list of constraints to enforce, or a domain-specific critic to call. There are many ways of choosing a refinement strategy at each iteration. Deciding the order to expand tasks is a small part of it. There is a clear need for devising new commitment strategies and experimentation. Most probably, there will not be a commitment strategy that works well in all planning problems. Controlled experiments with commitment strategies may serve to identify key features of planning problems, to help select the most efficient commitment strategy for a given problem.

The structure of HTN planning as refinement search is very suitable for parallel and distributed implementations. Such implementations can enable us to solve larger planning problems.

HTN planning can benefit from case-based planning techniques to guide the high-level search, to assess which branches in the search space are most likely to lead to solutions. It may also be possible to develop case-based reasoning techniques to act as domain-specific critics. There has been some recent work on sound and complete plan reuse in the state-based planning framework. Ways of accomplishing the same on HTN planning can be very helpful in providing speed up.

8.2.2 Expanding Capabilities of HTN Planning

In order to keep the basic HTN framework simple and easy to understand, context-dependent effects and quantified conditions are not allowed in the action representation. Furthermore, actions with durations and probabilistic effects are not addressed in this dissertation. Extending the representation for primitive tasks to cover these would be very useful.

Planning systems situated in real-world applications need to interact with a number of other software systems. For example, a planning system may need to access a flight database for schedules, or a geometric reasoner to analyze mechanical parts in a design. There is already a considerable body of work on database community for integrating heterogeneous information sources. Providing the same capability for planning systems would be extremely valuable.

In large scale planning applications, the planning domain may be so complicated that it may be infeasible to implement a single centralized planning system with expertise in every aspect of the problem. A distributed, agent-based approach may be more appropriate. Such a system would be composed of a number of agents, each agent expert in a small part of the problem, and controlling a small subset of the resources. Resolving interactions among agents and facilitating communication among agents is a critical part of such a system. The HTN language can be developed further to be the communication medium among agents, and the HTN constraint-handling techniques may be developed further to resolve conflicts among agents.

Classical AI planning systems usually work off-line. This is not suitable for dynamic domains with uncertainty, which require planning continuously and responding to unexpected situations. In such domains new tasks arrive continuously, while some tasks are being planned, and yet some others are being executed. Carrying the HTN planning techniques to an agent-based platform may also facilitate interleaving planning and execution.

Appendix A

Proofs of Theorems

Theorem 4 PLAN EXISTENCE is strictly semi-decidable, even if **P** is restricted to be propositional, to have at most two tasks in any task network, and to have only totally ordered methods.

Proof.

Membership: We can restate PLAN EXISTENCE as $\exists k\ sol_k(d, I, \mathcal{D}) \neq \emptyset$. Thus the problem is in Σ^1.

Hardness: Given two context-free grammars G_1 and G_2, whether $L(G_1) \cap L(G_2)$ is non-empty is an undecidable problem[32]. We define a reduction from this problem to PLAN EXISTENCE as follows:

Without loss of generality, assume both G_1 and G_2 have the same binary alphabet Σ, and they are in Chomsky normal form (at most two symbols at the right hand side of production rules). Refer to [32] to see how any context-free grammar can be converted into this form. Similarly, assume that the sets of non terminals Γ_1 and Γ_2 for each grammar are disjoint; i.e. $\Gamma_1 \cap \Gamma_2 = \emptyset$. We also assume neither language contains the empty string. It is easy to check whether a CFG derives empty string. If both languages contain the empty string, then their intersection is non-empty; we can simply return a simple HTN problem that has a solution. If one of the languages does not contain the empty string, it does not affect the intersection to remove the empty string from the other language.

It is quite easy to see that using methods we can simulate context-free grammars: Primitive task symbols mimic the terminals, compound task symbols mimic the non-terminals, and methods mimic the production rules. The difficulty is in making sure there is a string produced by both G_1 and G_2. We achieve this with the help of the constraints in methods.

For each terminal $a \in \Sigma$, we introduce a proposition p_a. We also need another proposition called *turn*.

Let the initial state $I = \{turn\}$.

For each terminal $a \in \Sigma$, we introduce two primitive tasks (one for each grammar) f_{a1} and f_{a2} such that f_{a1} has the preconditions $\{turn\}$ and effects $\{p_a, \neg turn\}$; f_{a2} has the preconditions $\{p_a, \neg turn\}$ and effects $\{\neg p_a, turn\}$.

Intuitively, f_{a1} produces p_a, and f_{a2} consumes p_a. The proposition *turn* ensures that we use these primitive tasks alternately.

For each non-terminal B in each grammar, we introduce a compound task symbol t_B.

For each production rule $R : A \rightarrow B_1 B_2$, we introduce a method

$$(t_A, [(n_1 : \alpha_1)(n_2 : \alpha_2)\ (n_1 \prec n_2)])$$

, where

$$\alpha_i = \begin{cases} perform[t_{Bi}] & \text{if } B_i \text{ is a nonterminal,} \\ do[f_{a1}] & \text{if } B_i \text{ is a terminal } a, \text{ and } R \text{ is a production rule of } G_1, \\ do[f_{a2}] & \text{if } B_i \text{ is a terminal } a, \text{ and } R \text{ is a production rule of } G_2. \end{cases}$$

The input task network contains the three tasks $perform[t_{S1}]$, $perform[t_{S2}]$, and $do[f_{last}]$, where S_1, S_2 are the starting symbols of the grammars G_1, G_2 respectively, and f_{last} is a primitive task with no effects. The constraint formula states that $t_{S1} \prec t_{last}$, and $t_{S2} \prec f_{last}$, and that *turn* needs to be true immediately before f_{last}. The last condition ensures that the last primitive task f_{pa} belongs to G_2.

The task decompositions mimic the production rules of the grammars. The proposition *turn* ensures that each grammar contributes a primitive action to any plan alternatively, and the conditions with propositions p_a ensure that whenever G_1 contributes a primitive task f_{a1}, G_2 has to contribute f_{a2}. Thus, there is a plan iff G_1 and G_2 have a common word in their corresponding languages.

■

Theorem 5 There are HTN planning domains that contain only totally ordered methods each with at most two tasks, for which PLAN EXISTENCE is strictly semi-decidable.

Proof: We construct a planning domain $\mathcal{D}$ and show that planning in this domain is semi-decidable, using a reduction from the intersection of context-free grammars problem.

Domain Description We use four primitive tasks $f_{a1}, f_{b1}, f_{a2}, f_{b2}$ (they are exactly the same primitive tasks used in the previous proof), and another dummy primitive task f_{dummy} with no effects or preconditions.

We have three propositions $p_a, p_b, turn$, and a predicate R(X,Y,Z), used for expressing production rules of the form $X \rightarrow YZ$.

We declare the following four operators that specify the effects of those tasks:

$$
\begin{array}{lll}
(operator\ f_{a1} & (pre : turn) & (post : p_a, \neg turn)) \\
(operator\ f_{a2} & (pre : \neg turn, p_a) & (post : \neg p_a, turn)) \\
(operator\ f_{b1} & (pre : turn) & (post : p_b, \neg turn)) \\
(operator\ f_{b2} & (pre : \neg turn, p_b) & (post : \neg p_b, turn)) \\
(operator\ f_{dummy} & (pre :) & (post :))
\end{array}
$$

We use five compound tasks $t(A1), t(A2), t(B1), t(B2), t(Dummy)$ which correspond to our primitive tasks.

We declare five methods describing how those compound tasks expand to their corresponding primitive tasks:

$$
\begin{array}{llll}
(t(v) & [(n : do[f_{a1}]) & (v = A1)]) \\
(t(v) & [(n : do[f_{b1}]) & (v = B1)]) \\
(t(v) & [(n : do[f_{a2}]) & (v = A2)]) \\
(t(v) & [(n : do[f_{b2}]) & (v = B2)]) \\
(t(v) & [(n : do[f_{dummy}]) & (v = Dummy)])
\end{array}
$$

We use a predicate $R(v, v_1, v_2)$ to encode grammar rules. We declare a final method :

$$(t(v) \quad [(n_1 : perform[t(v_1)])(n_2 : perform[t(v_2)]) \quad (n_1 \prec n_2) \wedge (n_1, R(v, v_1, v_2))])$$

Basicly, this method specifies that a task $t(X)$ can be expanded to $t(Y)t(Z)$ iff there is a production rule of the form $X \rightarrow YZ$. Thus we have a domain with 5 operators and 5 methods.

The reduction Given two context-free grammars, here is how we create the initial state and the input task-network.

Let $G_i =< \Sigma, \Gamma_i, R_i > i = 1, 2$ be two context-free grammars. Without loss of generality, assume $\Sigma = \{a, b\}, \Gamma_1 \cap \Gamma_2 = \emptyset$, the production rules are in Chomsky normal form (at most two symbols at right hand sides), and the grammars don't use the symbols $\{A1, A2, B1, B2, Dummy\}$

Here is the initial state:

- For each production rule of the form $X \rightarrow YZ$, we assert a predicate $R(X, Y, Z)$.
- For each production rule of the form $X \rightarrow a$ from grammar i, we assert a predicate $R(X, A_i, Dummy)$. We handle rules of the form $X \rightarrow b$, similarly.
- Finally, we assert *turn*.

The input task network to the planner contains the three tasks $t(S1)$, $t(S2)$, and f_{dummy}, where $S1, S2$ are the starting symbols of the grammars G_1, G_2 respectively. The constraint formula states that both $t(S1)$ and $t(S2)$ precede f_{dummy}, and that *turn* needs to be true immediately before f_{last}. The last condition ensures that the last primitive task belongs to G_2.

How it works: The construction is very similar to that of Theorem 4. In that construction, we introduced a method for each production rule. This time, we observe that all those methods had the same structure, so instead we use a single method with variables and an extra constraint $R(X, Y, Z)$ that makes sure that we can expand $t(X)$ to $t(Y)t(Z)$ only when we have the corresponding production rule. The sequence of actions $t(S_i)$ can expand to corresponds to the strings that can be derived from S_i. The effects of the actions and the conditions on them ensure that in any final plan the actions from S_1 and S_2 alternate, and that whenever S_1 contributes an action, it has to be followed by the corresponding action in S_2. Obviously, the reduction can be done in linear time.

■

Theorem 6 PLAN EXISTENCE is decidable if **P** has a k-level-mapping for some integer k.

Proof. When there exists a k-level-mapping, no task can be expanded to a depth more than k. Thus, whether a plan exists can be determined after a finite number of expansions. ■

Theorem 7 PLAN EXISTENCE is EXPSPACE-hard and in DOUBLE-EXPTIME if **P** is restricted to be totally ordered. PLAN EXISTENCE is PSPACE-hard and in EXPTIME if **P** is further restricted to be propositional.

Proof.

Membership: Here, we present an algorithm that runs in DOUBLE-EXPTIME, and solves the problem. In the propositional case, the number of atoms, the number of states etc. would go one level down, and thus, the same algorithm would solve the problem in EXPTIME.

The basic idea is this: for each ground task t, states s_I, s_F, and set of ground literals $L = \{l_1, \ldots, l_k\}$, we want to compute whether there exists a plan for t starting at s_I and ending at s_F while protecting the literals in L (i.e. without making them false). We store our partial results in a table with an entry for each tuple $\langle t, s_I, s_F, L\rangle$. An entry in the table has value either yes, no, or unknown.

Here is the algorithm:

1. Initialize all the entries in the table to unknown.
2. For each s_I, s_F, L and ground primitive task f_p, compute whether executing f_p at s_I results in s_F, and that all literals in L are true in both S_I and s_F. Insert the result in the table.
3. For each method $\langle t, (n_1 : \alpha_1) \ldots (n_k : \alpha_k), \phi\rangle$ and the input task network do:
   - Replace each constraint of the form (n_i, l, n_j) with $(n_i, l, n_{i+1}) \wedge (n_{i+1}, l, n_{i+2}) \wedge \ldots \wedge (n_{j-1}, l, n_j)$.

(For simplicity, we assume the label of a node reflects its position in the total order.)

- Apply de Morgan's rule so that negations come before only atomic constraints.

4. Go over all the entries $\langle t, s_I, s_F, L\rangle$ in the table with value unknown, doing the following:

   For all ground instances of methods for t $\langle t, (n_1 : \alpha_1) \ldots (n_k : \alpha_k), \phi\rangle$ do:

   For all $k+1$ tuples of states $(s_0, \ldots, s_k)$ do:

   For all expansions ϕ' of ϕ into conjuncts do:

   (a) for each conjunct of the form (n_i, l) or (l, n_{i+1}), check whether s_i satisfies l.

   (b) For each $i \leq k$, let L'_i be the set of literals l such that (n_i, l, n_{i+1}) is a conjunct.
   Check whether the entry for $\langle t_i, s_{i-1}, s_i, L_i\rangle$ is yes.

   (c) Check whether the variable binding constraints are satisfied

   (d) If all checks are OK, enter yes to the table for $\langle t, s_I, s_F, L\rangle$.

5. If step 4 modified the table, then goto step 4.

6. For all ground instances $\langle (n_1 : \alpha_1) \ldots (n_k : \alpha_k), \phi\rangle$ of the input task network do

   For all $k+1$ tuples of states $(s_0, \ldots, s_k)$ do

   For all expansions ϕ' of ϕ into conjuncts do

   (a) for each conjunct of the form (n_i, l) or (l, n_{i+1}), check whether s_i satisfies l.

   (b) For each $i \leq k$, let L'_i be the set of literals l such that (n_i, l, n_{i+1}) is a conjunct.
   Check whether the entry for $\langle t_i, s_{i-1}, s_i, L_i\rangle$ is yes.

   (c) Check whether the variable binding constraints are satisfied

   (d) If all checks are OK, halt with success; if not, halt with failure.

The algorithm works bottom-up. For all ground tasks, state pairs and protection sets $\langle t, s_I, s_F, L\rangle$, it computes whether there exists a plan for t starting at s_I and ending at s_F that does not violate the literals in L. When step 4 terminates without any modification to the table, the table contains all the answers. Thus in step 6, we can check whether the input task network can be achieved.

The table has a doubly exponential number of entries (roughly the cube of the number of states times number of ground tasks). Step 4 is executed at most a doubly exponential number of times (when we make one modification at each step). At each execution step 4 goes over all the entries in the table, taking double exponential time. Processing each entry takes double exponential time. The resultant time is the product of these, which is still

double exponential. The rest of the steps in the algorithm obviously do not take more than double exponential time. Thus the algorithm runs in double exponential time.

When we restrict the problem to be propositional, the number of states goes down from doubly exponential to exponential, and so does the size of table and the number of executions in all the steps. Thus in propositional case, the algorithm runs in exponential time.

Hardness: PLAN EXISTENCE, restricted to totally ordered regular planning domains, is a special case of our problem. But in Theorems 8 and 9, we prove that under this restricted version of PLAN EXISTENCE is EXPSPACE-hard (or PSPACE-hard in the propositional case). Thus the hardness follows.

■

Theorem 8 PLAN EXISTENCE is EXPSPACE-complete if **P** is restricted to be regular. It is still EXPSPACE-complete if **P** is further restricted to be totally ordered, with at most one non-primitive task symbol in the planning language, and all task networks containing at most two tasks.

Proof.

Membership: It suffices to present a nondeterministic algorithm that uses at most exponential space and solves the problem, as EXPSPACE=N-EXPSPACE, so that is what we will do. Since all task networks will contain at most one non-primitive task, all we need to do is keep track of what atoms need to be true/false immediately before, immediately after, and along that single task. Since there are an exponential number of atoms, we can do this within exponential space. Here is the algorithm:

1. Let d be the input task network.

2. If d contains only primitive tasks, then
   non-deterministically guess a total-ordering and variable-binding.
   If it satisfies the constraint formula and the preconditions of the primitive tasks, then halt with success; if not, halt with failure.

3. Non-deterministically, guess a total-ordering and variable-binding. The task network will be of the form

   $$[(n_1 : do[f_1]) \ldots (n_m : do[f_m])(n : perform[t])(n'_1 : do[f'_1]) \ldots (n'_u : do[f'_u])]$$

   with ordering $n_1 \prec n_2 \prec \ldots \prec n_m \prec n \prec n'_1 \prec \ldots \prec n'_u$.

   Note that t is the only non-primitive task in the network. Let s_i, s'_i, s_t be the states immediately after f_i, f'_i, t, respectively.

4. Eliminate all variable binding and task ordering constraints from the constraint formula using the guess in step 3.

5. Replace any constraint of the form (n_i, l, n'_j) with $(n_i, l, n_m) \wedge (n_m, l, n'_1) \wedge (n'_1, l, n'_j)$.

6. Replace any constraint of the form (n_i, l, n_j) or (n'_i, l, n'_j) with $(n_i, l) \wedge \ldots \wedge (n_{j-1}, l)$ and $(n'_i, l) \wedge \ldots \wedge (n'_{j-1}, l)$, respectively.

7. Process the constraint formula (using De Morgan's rule) so that negations apply to only atomic constraints.

8. Now the resultant constraint formula contains only conjuncts and disjuncts. For each disjunct, nondeterministically pick a component, obtaining a constraint formula containing only conjuncts.

9. Compute all the intermediate states before n and verify that all constraints of the form $(n_i, l), (l, n_i)$ are satisfied. Remove these constraints from the constraint formula.

10. For all the state constraints after n, use regression to determine what needs to be true immediately after n for those constraints to be satisfied[1].

11. Set the initial state I to $f_m(f_{m-1}(\ldots f_1(I)\ldots))$, i.e., the state that results from applying all the primitive tasks before t.

12. Now we can get rid of all the primitive tasks in the task network. The constraint formula contains only what needs to be true while we achieve t and what needs to be true immediately after we achieve t.

13. Nondeterministically, choose a method for t, expand it, and assign the resulting task network to d.

14. Go to step 2.

Hardness: In [22], we showed that plan existence problem in STRIPS representation is EXPSPACE-complete. Here we define a reduction from that problem.

The plan existence problem in STRIPS representation is defined as "Given a set of constants, a set of predicates, a set of STRIPS operators, an initial state and a goal, is there a plan that achieves the goal?"

Given such a problem, we transform it into an HTN planning problem as follows:

We use the same set of constants and predicates. We will also have the same initial state. For each STRIPS operator o, we define a primitive task f_o that has exactly the same preconditions and postconditions as o. We also need an extra primitive task f_ϵ with no effects, that will be used as a dummy.

We will need a single non-primitive task t that can be expanded to any executable sequence of actions. Thus we declare the following methods for t:

[1]Here is how we do this. Consider the task $(n'_i : t'_i)$ and the condition (n'_i, l). When we regress this condition, we get $TRUE$ if t'_i asserts l, $FALSE$ if t'_i denies l, or (n'_{i-1}, l) if t'_i neither denies nor asserts l. This is what needs to be $TRUE$ before t'_i, for the condition to hold after t'_i.

- $(t,\ [(n_1 : f_\epsilon), TRUE])$;
- for each STRIPS operator o

$$(t,\ [(n_1 : f_o)(n_2 : t), (n_1 \prec n_2)]).$$

Finally, the input task network will be of the form

$$\langle (n_1 : t), (n_1, g_1) \wedge (n_1, g_2) \ldots \wedge (n_1, g_k)) \rangle,$$

where g_i are the goals of the STRIPS problem.

The resulting HTN problem satisfies all the restrictions of the theorem. A plan σ solves the STRIPS plan existence problem instance iff σ is a solution for the HTN planning problem instance. Hence the reduction is correct. Obviously, the reduction is in polynomial time. ■

Theorem 9 PLAN EXISTENCE is PSPACE-complete if **P** is restricted to be regular and propositional. It is still PSPACE-complete if **P** is further restricted to be totally ordered, with at most one non-primitive task symbol in the planning language, and all task networks containing at most two tasks.

Proof.

Membership: The algorithm we presented for the membership proof in Theorem 8 works also for the propositional case. In the propositional case we have only a linear number of atoms, and as a result, the size of any state is polynomial. Thus the algorithm requires only polynomial space.

Hardness: In [12, 22], it is shown that plan existence problem in STRIPS representation is PSPACE-complete if it is restricted to be propositional. The reduction from STRIPS style planning that we presented in the hardness proof of Theorem 8 also works for the propositional case. ■

Theorem 10 If **P** is restricted to be regular and $\mathcal{D}$ is fixed in advance, then PLAN EXISTENCE is in PSPACE. Furthermore, there exists fixed regular HTN planning domains $\mathcal{D}$ for which PLAN EXISTENCE is PSPACE-complete.

Proof. When $\mathcal{D}$ is fixed in advance, the number of ground instances of predicates and tasks will be polynomial in the size of the input. Thus we can reduce it to propositional regular HTN-planning. As a direct consequence of Theorem 9, it is in PSPACE.

In the proof of Theorem 5.17 in [22], we had presented a set of STRIPS operators containing variables for which planning is PSPACE-complete. Applying the reduction defined in the hardness proof of Theorem 8 to this set of operators would give a regular HTN planning domain (containing variables) with the same set of solutions and complexity as its STRIPS counterpart. ■

Theorem 11 PLAN EXISTENCE is NP-complete if **P** is restricted to be primitive, or primitive and totally ordered, or primitive and propositional. However, PLAN EXISTENCE is in polynomial time **P** is restricted to be primitive, totally ordered, and propositional.

Proof. If **P** is primitive, propositional and totally ordered, we can compute whether an atomic constraint is satisfied in linear time. In order to find a plan, all we need to do is to check whether the constraint formula is satisfied, which can be done in polynomial time.

Membership: Given a primitive task network, we can nondeterministically guess a total ordering and variable binding, and then we can verify that it satisfies the constraint formula in polynomial time. Thus the problem is always in NP.

Hardness: There are three cases:

Case 1: Primitive and propositional.

We define a reduction from satisfiability problem as follows:

Given a boolean formula, we define a planning problem such that it uses the same set of propositions as the boolean formula. we have two primitive tasks for each proposition, one that deletes the proposition, and one that adds the proposition. The initial state is empty. The input task network contains all the primitive tasks, and the constraint formula states that the boolean formula needs to be true in the final state.

If there exists a total ordering that satisfies the constraint formula, the truth values of propositions in the final state would satisfy the boolean formula; if there is a truth assignment that satisfies the boolean formula, we can order the tasks such that if a proposition p is assigned true, then the primitive task that adds it is ordered after the primitive task that deletes it (and vice versa), coming up with a plan that achieves the task network.

Obviously, the reduction can be done in linear time.

Case 2: Primitive and totally ordered.

Again, we define a reduction from satisfiability problem.

We will use two constant symbols t and f, standing for true and false, respectively.

For each proposition p in the boolean formula, we will introduce a unary predicate P, and a unary primitive task symbol $T_p(v_p)$ that has the effect $P(v_p)$.

The initial state will be empty, and the input task network will contain one task for each proposition, namely $T_p(v_p)$.

We construct a formula F from the input boolean formula by replacing each proposition p with $P(t)$. Our constraint formula will require F to be true in the final state.

If there exists a truth assignment that satisfies the boolean formula, we can construct a variable binding such that v_p is bound to t whenever p is assigned true, and v_p is bound to f otherwise. This variable binding would make sure the constraints are satisfied.

If there exists a variable binding that achieves the task network, we construct the following truth assignment that satisfies the boolean formula. We assign true to p iff v_p is bound to t.

Obviously, the reduction can be done in polynomial time.

Case 3: Primitive.

Both case 1 and case 2 are special cases of case 3, so hardness follows immediately.

■

Appendix B

UM-Translog Domain Specification

B.1 Symbol Declarations

```
;


(VARIABLES
    ?v ?p ?l ?c ?r ?ol ?dl ?t
    ?package ?origin ?destination ?location ?region1 ?region2
    ?hub ?ocity ?dcity ?city1 ?city2 ?cityh ?samecity ?tcenter
    ?tcenter1 ?tcenter2 ?tt ?vehicle ?tc ?train ?ptype ?vtype ?route
    ?r ?rtype ?vptype ?type1 )

(CONSTANTS
      air-route mail perishable refrigerated granular airplane
      armored auto bulky cars city1 city1-cl1 city1-cl2 city1-ts1
      city1-ts2 city1-ap1 city1-ap2 city2 city2-cl1 city2-ap1
      city2-ts1 region1 region2 city3 city3-cl1 city3-ap1 city3-ts1
      region1-ap1 region1-ts1 rail-route-1 rail-route-2 rail-route-3
      rail-route-4 road-route-1 road-route-2 air-route-1 air-route-2
      air-route-3 air-route-4 ramp1a ramp1b ramp2 ramp3 ramp4
      road-route-i1547 road-route-i1548 pkg-1 truck-1 not-tcenter
      not-hub train-station tcenter hub not-hazardous not-traincar
      clocation tcenter airport train-station city region crane
      flatbed hazardous hopper hub livestock liquid local-road-route
      plane-ramp rail-route air-route regular road-route tanker
      train traincar truck tcenter valuable )

(PRIMITIVE-TASKS
      affix-warning-signs attach-conveyor-ramp
attach-train-car close-door close-valve collect-fees
collect-insurance connect-chute connect-hose
decontaminate-interior deliver detach-conveyor-ramp
detach-train-car disconnect-chute disconnect-hose
do-clean-interior do-nothing empty-hopper empty-tank
fill-hopper fill-tank fill-trough load-cars load-livestock
load-package lower-ramp move-vehicle obtain-permit open-door
open-valve pick-up-package-ground pick-up-package-vehicle
post-guard-inside post-guard-outside put-down-package-ground
put-down-package-vehicle raise-ramp remove-guard
remove-warning-signs unload-cars unload-livestock
unload-package )


(PREDICATES
      (at-equipment 15) (at-package 100) (at-vehicle 40)
available chute-connected clean-interior connects
decontaminated-interior door-open empty fees-collected
guard-inside guard-outside have-permit hose-connected in-city
in-region insured pc-compatible pv-compatible ramp-available
ramp-connected ramp-down rv-compatible serves trough-full type
valve-open warning-signs-affixed
)


(COMPOUND-TASKS
      (carry 79) (carry-between-tcenters 75) (carry-direct 70)
```

```
      (carry-via-hub 78) handle-hazardous handle-insurance
      (load-top 25) (unload-top 25) (load-haz 22) (load-val 22)
      (unload-haz 22) (unload-val 22) (load 20) pickup
(transport 90)(unload 20)
)

,
```

B.2 Actions

```
;

(operator open-door(?v)
   post(
            (door-open ?v))
)


(operator close-door(?v)
   post(
 (~door-open ?v))
)

;————————————————————————

(operator load-package(?p ?v ?l)
   pre (
  (at-package ?p ?l)
  (at-vehicle ?v ?l))
   post(
            (at-package ?p ?v)
  (~at-package ?p ?l))
)

(operator unload-package(?p ?v ?l)
   pre (
 (at-package ?p ?v)
 (at-vehicle ?v ?l)  )
   .post(
 (at-package ?p ?l)
 (~at-package ?p ?v))
)

,————————————————————————

(operator pick-up-package-ground(?p ?c ?l)
   :pre (
  (empty ?c)
  (at-equipment ?c ?l)
  (at-package   ?p ?l)  )
   .post(
            (at-package ?p ?c)
  (~empty ?c)
  (~at-package ?p ?l))
)

(operator put-down-package-ground(?p ?c ?l)
   .pre (
  (at-equipment ?c ?l)
  (at-package   ?p ?c)  )
   ·post(
            (empty ?c)
  (at-package ?p ?l)
  (~at-package ?p ?c))
)


(operator pick-up-package-vehicle(?p ?c ?v ?l)
    pre (
(empty ?c)
(at-equipment ?c ?l)
(at-package ?p ?v)
(at-vehicle ?v ?l)    )
    post(
     (at-package ?p ?c)
     (~empty ?c)
     (~at-package ?p ?v))
)

(operator put-down-package-vehicle(?p ?c ?v ?l)
   .pre (
(at-package ?p ?c)
(at-equipment ?c ?l)
(at-vehicle ?v ?l)    )
```

```
   .post(
    (empty ?c)
    (at-package ?p ?v)
    (~at-package ?p ?c))
)

;--------------------------------------

(operator connect-chute(?v)
  :post(
   (chute-connected ?v))
)

(operator disconnect-chute(?v)
   :post(
    (~chute-connected ?v))
)

;--------------------------------------

(operator fill-hopper(?p ?v ?l)
   :pre (
(chute-connected ?v)
(at-vehicle ?v ?l)
(at-package ?p ?l)    )
   :post(
    (at-package ?p ?v)
    (~at-package ?p ?l))
)

(operator empty-hopper(?p ?v ?l)
   pre (
(chute-connected ?v)
(at-vehicle ?v ?l)
(at-package ?p ?v)    )
   post(
    (at-package ?p ?l)
    (~at-package ?p ?v))
)

;--------------------------------------

(operator raise-ramp(?v)
   :post(
 (~ramp-down ?v))
)

(operator lower-ramp(?v)
   :post(
  (ramp-down ?v))
)

;--------------------------------------

(operator fill-trough(?v)
   :post(
   (trough-full ?v))
)

;--------------------------------------

(operator load-livestock(?p ?v ?l)
   :pre (
(at-package ?p ?l)
(at-vehicle ?v ?l)
(ramp-down ?v)      )
   post(
    (at-package ?p ?v)
    (~at-package ?p ?l)
    (~clean-interior ?v))
)

(operator unload-livestock(?p ?v ?l)
   :pre (
(at-package ?p ?v)
(at-vehicle ?v ?l)
(ramp-down ?v)        )
   :post(
    (at-package ?p ?l)
    (~at-package ?p ?v)
    (~trough-full ?v))
)

;--------------------------------------

(operator do-clean-interior(?v)
   :post(  (clean-interior ?v))
```

```
)

;-----------------------------

(operator load-cars(?p ?v ?l)
 pre (
                (at-package ?p ?l)
                (at-vehicle ?v ?l)
                (ramp-down ?v)      )
.post(
        (at-package ?p ?v)
    (~at-package ?p ?l))
)

(operator unload-cars(?p ?v ?l)
:pre (
    (at-package ?p ?l)
    (at-vehicle ?v ?l)
    (ramp-down ?v)      )
:post(
        (at-package ?p ?l)
  (~at-package ?p ?v))
)

,-----------------------------

(operator connect-hose(?v)
   :post(  (hose-connected ?v))
)

(operator disconnect-hose(?v ?p)
   :pre ( (hose-connected ?v))
   :post( (~hose-connected ?v))
)

,-----------------------------

(operator open-valve(?v)
   :post(  (valve-open ?v)    )
)

(operator close-valve(?v)
    post(  (~valve-open ?v)   )
)

,-----------------------------

(operator fill-tank(?v ?p ?l)
   :pre (
 (hose-connected ?v)
 (valve-open ?v)
 (at-package ?p ?l)      )
   .post(
    (at-package ?p ?v)
    (~at-package ?p ?l))
)

(operator empty-tank(?v ?p ?l)
    pre (
(hose-connected ?v)
(valve-open ?v)
(at-package ?p ?v)      )
   ·post(
    (at-package ?p ?l)
    (~at-package ?p ?v))
)

,-----------------------------

(operator move-vehicle(?v ?ol ?dl ?r)
   .pre (  (at-vehicle ?v ?ol)  )
   :post(
                (at-vehicle ?v ?dl)
    (~at-vehicle ?v ?ol))
)

;-----------------------------

(operator attach-train-car(?t ?c ?l)
   ·pre (
 (at-vehicle ?c ?l)
 (at-vehicle ?t ?l)  )
    post(
                (at-vehicle ?c ?t)
    (~at-vehicle ?c ?l))
)
```

```
(operator detach-train-car(?t ?c ?l)
  :pre (
(at-vehicle ?t ?l)
(at-vehicle ?c ?t) )
  .post(
   (at-vehicle ?c ?l)
   (~at-vehicle ?c ?t))
)

;-------------------------

(operator attach-conveyor-ramp(?v ?r ?l)
   pre (
(ramp-available ?r)
(at-equipment ?r ?l)
(at-vehicle ?v ?l)   )
  :post(
                (ramp-connected ?r ?v)
   (~ramp-available ?r))
)

(operator detach-conveyor-ramp(?v ?r ?l)
  :pre (
                    (ramp-connected ?r ?v)
(at-equipment ?r ?l)
(at-vehicle ?v ?l)   )
  ·post(
   ( ramp-available ?r)
   (~ramp-connected ?r ?v))
)

;-----------

(operator affix-warning-signs (?v)
  ·post( (warning-signs-affixed ?v))
)

(operator remove-warning-signs (?v)
  :post( (~warning-signs-affixed ?v))
)

;-----------

(operator post-guard-outside(?v)
  :post( (guard-outside ?v)
  (~guard-inside ?v))
)

(operator post-guard-inside(?v)
  :post( (guard-inside ?v)
  (~guard-outside ?v))
)

(operator remove-guard(?v)
   post(   (~guard-outside ?v)
    (~guard-inside ?v))
)

;-----------

(operator decontaminate-interior(?v)
  :post( (decontaminated-interior ?v))
)


(operator obtain-permit(?p)
  :post( (have-permit ?p))
)

(operator collect-fees(?p)
  .post( (fees-collected ?p))
)

(operator collect-insurance(?p)
  :post( (insured ?p))
)

(operator deliver(?p)
  ·post(   (~have-permit ?p)
   (~fees-collected ?p)
   (~insured ?p))
)


;
```

B.3 Methods

```
;

(variables ?package
  ?origin ?destination  ?location
  ?ocity ?dcity ?city1 ?city2 ?samecity
  ?tcenter ?tcenter1 ?tcenter2 ?tt
  ?region1 ?region2 ?hub
  ?vehicle  ?tc ?train
  ?ptype ?vtype ?route ?r ?rtype ?vptype ?type1

)


;;; top-level declare-method: used for top-level goal
(declare-method AT-PACKAGE(?package ?destination)
   .expansion (
           (n1 TRANSPORT ?package ?origin ?destination)
          )
   :formula (and
      (initially (AT-PACKAGE ?package ?origin) )
           (after  (AT-PACKAGE ?package ?destination) n1))
)


(declare-method TRANSPORT(?package ?origin ?destination)
       :expansion (
    (n1  PICKUP  ?package)
     (n2  CARRY   ?package ?origin ?destination)
     (n3  DELIVER ?package)
                )
       :formula (and (ord n1 n2) (ord n2 n3))
,       effects (
 ;              (n2 :delete (AT-PACKAGE ?package ?origin))
  ,             (n2 :assert (AT-PACKAGE ?package ?destination))
   ,            )
 )
;-------------
, PICKUP DECLARE-METHODS
;-------------

(declare-method PICKUP(?package)
       expansion (
              (n1  COLLECT-FEES    ?package)
              (n2  HANDLE-INSURANCE ?package)
              (n3  HANDLE-HAZARDOUS ?package))
)

;-------------

(declare-method HANDLE-INSURANCE(?package)
        expansion (
                (n1  COLLECT-INSURANCE ?package)
               )
       :formula (initially (TYPE ?package VALUABLE))
)

(declare-method HANDLE-INSURANCE(?package)
        expansion (
                (n1 DO-NOTHING)
               )
       :formula (initially (~TYPE ?package VALUABLE))
)

,-------------

(declare-method HANDLE-HAZARDOUS(?package)
       .expansion (
                (n1  OBTAIN-PERMIT ?package)
               )
        formula (initially (TYPE ?package HAZARDOUS) n1)
)

(declare-method HANDLE-HAZARDOUS(?package)
        expansion (
                (n1 DO-NOTHING)
               )
       :formula (initially (~TYPE ?package HAZARDOUS) )
)


;-------------
;  CARRY DECLARE-METHODS - TOP LEVEL   ** THESE BIND TCENTERS **
,-------------

,,; top-level carry declare-method 1
```

```
(declare-method CARRY(?package ?origin ?destination)
        :expansion (
                (n1 CARRY-DIRECT ?package ?origin ?destination)
                )
;       :effects (
 ;              (n1 :delete (AT-PACKAGE ?package ?origin))
  ,             (n1 :assert (AT-PACKAGE ?package ?destination))
   ,            )
)

;;; 2 carry:
;;;?origin (tcenter, not hub) -¿ ?destination (tcenter, not hub)
(declare-method CARRY(?package  ?origin  ?destination)
         expansion(
                (n1  CARRY-VIA-HUB ?package ?origin ?destination)
                )
,       :effects (
 ;              (n1 :delete (AT-PACKAGE ?package ?origin))
  ,             (n1 :assert (AT-PACKAGE ?package ?destination))
   ,            )

        :formula (and
                  (initially (IN-CITY ?origin      ?ocity))
                  (initially (IN-CITY ?destination ?dcity))
                  (not (veq ?ocity ?dcity))
                  (initially (TYPE ?origin      TCENTER))
                  (initially (TYPE ?destination TCENTER))
                  (initially (~TYPE ?origin   HUB  ))
                  (initially (~TYPE ?destination HUB))
                  (initially (AVAILABLE ?origin))
                  (initially (AVAILABLE ?destination))
                 )
)

;;; 3 carry.
;;;?origin (not tcenter) -¿ ?tcenter -¿ ?destination (tcenter)
(declare-method CARRY(?package ?origin ?destination)
         expansion (
     (n0  CARRY-DIRECT ?package ?origin ?tcenter)
                  (n1  CARRY-BETWEEN-TCENTERS
   ?package ?tcenter ?destination)
                  )
;        effects (
 ;              (s0 .delete (AT-PACKAGE ?package ?origin))
  ;             (s0 :assert (AT-PACKAGE ?package ?tcenter))
   ;            (n1 :delete (AT-PACKAGE ?package ?tcenter))
    ;           (n1 :assert (AT-PACKAGE ?package ?destination))
     ,          )
        :formula (and
            (ord n0 n1)
                  (initially (~TYPE ?origin      TCENTER) )
                  (initially (TYPE ?destination TCENTER)  )
                  (initially (IN-CITY ?origin ?ocity)     )
                  (initially (IN-CITY ?destination ?dcity))
                  (not (veq ?ocity ?dcity))
                  (initially (SERVES ?tcenter ?ocity) )
                  (initially (TYPE ?tcenter TCENTER) )
            (not (veq ?tcenter ?destination) )
                  (initially (TYPE ?tcenter ?tt) )
                  (not (veq ?tt TCENTER))
                  (initially (TYPE ?destination ?tt))
                  (initially (AVAILABLE ?tcenter)  )
                  (initially (AVAILABLE ?destination))
                 )
)

;;; 4 carry:
;;; ?origin (tcenter) -¿ ?tcenter -¿ ?destination (not tcenter)
(declare-method CARRY(?package ?origin ?destination)
        .expansion (
     (n1  CARRY-BETWEEN-TCENTERS ?package  ?origin ?tcenter)
     (n2  CARRY-DIRECT ?package  ?tcenter ?destination)
                  )
,        effects (
 ;              (n1 :delete (AT-PACKAGE ?package ?origin))
  ,             (n1 :assert (AT-PACKAGE ?package ?tcenter))
   ,            (n2 :delete (AT-PACKAGE ?package ?tcenter))
    ,           (n2 :assert (AT-PACKAGE ?package ?destination))
     ,          )
        .formula (and
                  (ord n1 n2)
                  (initially (TYPE ?origin      TCENTER)     )
                  (initially (~TYPE ?destination TCENTER)  )
                  (initially (IN-CITY ?origin ?ocity)      )
                  (initially (IN-CITY ?destination ?dcity)  )
                  (not (veq ?ocity ?dcity))
                  (initially (SERVES ?tcenter ?dcity)  )
```

```
          (initially (TYPE ?tcenter TCENTER) )
                (initially (TYPE ?tcenter ?tt) )
                (not (veq ?tt TCENTER))
                (initially (TYPE ?origin ?tt) )
                (not (veq ?tcenter ?origin))
                (initially (AVAILABLE ?origin)      )
                (initially (AVAILABLE ?tcenter)     )
               )
)

;;; carry
;; ?origin (not tcenter) -¿
;;     ?tcenter1 -¿ ?tcenter2 -¿ ?destination (not tcenter)
(declare-method CARRY(?package  ?origin  ?destination)
       :expansion (
                (n0  CARRY-DIRECT ?package  ?origin   ?tcenter1)
                (n1  CARRY-BETWEEN-TCENTERS
                    ?package  ?tcenter1 ?tcenter2)
                (n2  CARRY-DIRECT
                    ?package  ?tcenter2 ?destination)
               )
,       .effects (
 ,              (n1 :delete (AT-PACKAGE ?package ?tcenter1))
  ,             (n1  assert (AT-PACKAGE ?package ?tcenter2))
   ,            (n0 ·delete (AT-PACKAGE ?package ?origin))
    ,           (n0  assert (AT-PACKAGE ?package ?tcenter1))
     ,          (n2  assert (AT-PACKAGE ?package ?destination))
      ;         (n2 :delete (AT-PACKAGE ?package ?tcenter2)))
       :formula (and
                 (ord n0 n1)
          (ord n1 n2)
                (initially (~TYPE ?origin     TCENTER) )
                (initially (~TYPE ?destination TCENTER) )
                (initially (IN-CITY ?origin ?ocity)      )
                (initially (IN-CITY ?destination ?dcity) )
                (not (veq ?ocity ?dcity))
                (initially (SERVES ?tcenter1 ?ocity) )
                (initially (SERVES ?tcenter2 ?dcity) )
          (initially (TYPE ?tcenter1 TCENTER) )
                (initially (TYPE ?tcenter1 ?tt)     )
                (not (veq ?tt TCENTER))
                (initially (TYPE ?tcenter2 ?tt) )
          (initially (TYPE ?tcenter2 TCENTER) )
                (initially (AVAILABLE ?tcenter1) )
                (initially (AVAILABLE ?tcenter2) )
               )
)

;————————————
; CARRY DECLARE-METHODS - BETWEEN TCENTERS.
;————————————

;;; carry: ?tcenter1 -¿ ?tcenter2 where ?tcenter1 = ?tcenter2
(declare-method CARRY-BETWEEN-TCENTERS(?package ?tcenter1 ?tcenter2)
  :expansion (
     (n1 DO-NOTHING))
,       effects (
 ,              (n1  delete (AT-PACKAGE ?package ?tcenter1))
  ,             (n1 :assert (AT-PACKAGE ?package ?tcenter2)))
       .formula (veq ?tcenter1 ?tcenter2)
)

;;; carry: ?tcenter1 -¿ ?tcenter2
(declare-method CARRY-BETWEEN-TCENTERS(?package ?tcenter1 ?tcenter2)
  expansion (
     (n1  CARRY-DIRECT ?package ?tcenter1 ?tcenter2)
               )
 ,      .effects (
  ,             (n1 ·delete (AT-PACKAGE ?package ?tcenter1))
   ;            (n1  assert (AT-PACKAGE ?package ?tcenter2)))
       .formula (and
                 (not (veq ?tcenter1 ?tcenter2))
                 (initially (TYPE ?tcenter1 ?type1) )
                 (not (veq ?type1 TCENTER))
                 (not (veq ?type1 HUB))
                 (initially (TYPE ?tcenter2 ?type1) )
                )
)

;;; carry. ?tcenter1 (not hub) -¿ ?tcenter2 (not hub)
(declare-method CARRY-BETWEEN-TCENTERS(?package ?tcenter1 ?tcenter2)
  expansion (
     (n1  CARRY-VIA-HUB ?package  ?tcenter1 ?tcenter2)
               )
 ,      ·effects (
  ,             (n1 .delete (AT-PACKAGE ?package ?tcenter1))
   ;            (n1 :assert (AT-PACKAGE ?package ?tcenter2)))
```

```
 :formula (and
     (initially (~TYPE ?tcenter1 HUB) )
     (initially (~TYPE ?tcenter2 HUB) )
                 (not (veq ?tcenter1 ?tcenter2))
                )
)


;;; carry  ?tcenter1 -¿ ?hub -¿ ?tcenter2.
(declare-method CARRY-VIA-HUB(?package ?tcenter1 ?tcenter2)
  ·expansion (
     (n1  CARRY-DIRECT ?package ?tcenter1 ?hub)
     (n2  CARRY-DIRECT ?package ?hub ?tcenter2)
                 )
;         .effects (
 ;               (n1 :assert (AT-PACKAGE ?package ?hub) )
  ;              (n1 :delete (AT-PACKAGE ?package ?tcenter1))
   ;             (n2 ·assert (AT~PACKAGE ?package ?tcenter2))
    ;            (n2 :delete (AT-PACKAGE ?package ?tcenter1)))
  ·formula (and
            (ord n1 n2)
                 (initially (~TYPE ?package HAZARDOUS)  )
                 (initially (IN-CITY ?tcenter1 ?city1)  )
                 (initially (IN-CITY ?tcenter2 ?city2)  )
                 (initially (IN-REGION ?city1 ?region1) )
                 (initially (IN-REGION ?city2 ?region2) )
                 (initially (SERVES ?hub ?region1)  )
                 (initially (SERVES ?hub ?region2)  )
                 (initially (AVAILABLE ?hub)  )
                )
)


;;; carry. ?tcenter1 -¿ ?hub -¿ ?tcenter2.
(declare-method CARRY-VIA-HUB(?package ?tcenter1 ?tcenter2)
   expansion (
     (n1  CARRY-DIRECT ?package ?tcenter1 ?hub)
     (n2  CARRY-DIRECT ?package ?hub ?tcenter2)
                 )
,         .effects (
 ,               (n1 ·assert (AT-PACKAGE ?package ?hub) )
  ;              (n1 :delete (AT-PACKAGE ?package ?tcenter1))
   ;             (n2 :assert (AT~PACKAGE ?package ?tcenter2))
    ,            (n2 :delete (AT-PACKAGE ?package ?tcenter1)))
  :formula (and
      (ord n1 n2)
                  (initially (TYPE ?package HAZARDOUS)  )
                  (initially (PC-COMPATIBLE ?cityh HAZARDOUS)  )
                  (initially (IN-CITY ?tcenter1 ?city1)  )
                  (initially (IN-CITY ?tcenter2 ?city2)  )
                  (initially (IN-REGION ?city1 ?region1)  )
                  (initially (IN-REGION ?city2 ?region2)  )
                  (initially (SERVES ?hub ?region1)  )
                  (initially (SERVES ?hub ?region2)  )
                  (initially (IN-CITY ?hub ?cityh)  )
                  (initially (AVAILABLE ?hub)  )
                 )
)
;-------------
;  CARRY DECLARE-METHODS - DIRECT.
'-------------

(declare-method CARRY-DIRECT(?package ?origin ?destination)
         .expansion (
                  (n1  AT-VEHICLE ?vehicle ?origin)
                  (n2  LOAD-top   ?package ?vehicle ?origin)
                  (n3  AT-VEHICLE ?vehicle ?destination)
                  (n4  UNLOAD-top ?package ?vehicle ?destination)
                 )
         :formula(and
    (ord n1 n2)
          (ord n2 n3)
    (ord n3 n4)
    (before (AT-PACKAGE ?package ?origin)  n2)
                 (initially (TYPE ?package ?ptype)  )
                 (between  (AT-VEHICLE ?vehicle  ?origin)  n1 n2)
                 (between  (AT-PACKAGE ?package ?vehicle)  n2 n4)
                 (initially (AVAILABLE ?vehicle))
                 (initially (TYPE ?vehicle ?vtype)  )
                 (initially (PV-COMPATIBLE ?ptype ?vtype)  )
                 (or
          (and  , same city via truck
   (initially (TYPE ?vehicle TRUCK)  )
   (initially (IN-CITY ?origin ?ocity)  )
   (initially (IN-CITY ?destination ?ocity)))
  (and  ;diff city via truck
   (initially (TYPE ?vehicle TRUCK)  )
```

```
  (initially (IN-CITY ?origin ?ocity) )
  (initially (IN-CITY ?destination ?dcity))
  (initially (CONNECTS
     ?route ROAD-ROUTE
     ?ocity  ?dcity))
  (initially (AVAILABLE ?route)   ))
 (and  ,not truck
  (initially (~TYPE ?vehicle TRUCK) )
  (initially (CONNECTS
     ?route ?rtype
     ?origin ?destination))
  (initially (RV-COMPATIBLE ?rtype ?vhtype))
  (initially (TYPE ?vehicle ?vhtype) )
  (initially (AVAILABLE ?route) )))
                )
)


;-----------
; MOVE VEHICLE DECLARE-METHODS
;-----------

;;; move: ?origin -¿ ?location, via road
(declare-method AT-VEHICLE(?vehicle ?location)
    expansion(
      (n1  MOVE-VEHICLE ?vehicle  ?origin ?location ?r)
      )
   :formula (and
     (before (AT-VEHICLE ?vehicle  ?origin)  n1)
     (initially (TYPE ?vehicle TRUCK) )
     (initially (IN-CITY ?origin ?ocity) )
     (or
      (and ; same city
(veq ?r LOCAL-ROAD-ROUTE)
(initially (IN-CITY ?location ?ocity) ))
      (and ;different city
(initially (IN-CITY ?location ?dcity) )
(initially (CONNECTS ?r ROAD-ROUTE ?ocity ?dcity) ))
      ))
)


;;; move: ?origin -¿ ?location
(declare-method AT-VEHICLE(?vehicle ?location)
    expansion (
(n1  MOVE-VEHICLE ?vehicle  ?origin ?location ?r)
)
   .formula (and
     (initially (~TYPE ?vehicle TRAINCAR) )
     (before (AT-VEHICLE ?vehicle  ?origin)  n1)
     (initially (TYPE ?vehicle ?vtype) )
     (initially (CONNECTS ?r ?rtype  ?origin ?location) )
     (initially (RV-COMPATIBLE ?rtype ?vtype) )
     )
   )

;---------
; TRAIN CAR DECLARE-METHODS
;---------

;;; send train to pickup car, deliver it to destination, and detach it
(declare-method AT-VEHICLE(?tc ?destination)
    expansion (
(n1  AT-VEHICLE  ?train ?origin)
(n2  ATTACH-TRAIN-CAR ?train ?tc ?origin)
(n3  AT-VEHICLE ?train ?destination)
(n4  DETACH-TRAIN-CAR ?train ?tc ?destination)
)
   .formula (and
     (ord n1 n2)
     (ord n2 n3)
     (ord n3 n4)
     (initially (TYPE ?tc  TRAINCAR) )
     (before (AT-VEHICLE ?tc ?origin)  n2)
     (initially (CONNECTS ?r RAIL-ROUTE ?origin ?destination))
     (initially (TYPE ?train TRAIN) )
     (between (AT-VEHICLE ?tc ?train )  n2 n4)
     (between (AT-VEHICLE ?train ?origin)  n1 n2)
     (between (AT-VEHICLE ?train ?destination)  n3 n4 )
     )
)

,

     ,
```

```
(variables ?p ?v ?l ?c ?r)

(declare-method load-top (?p ?v ?l)
  :expansion(
    (n1 load-haz ?p ?v ?l)
    (n2 load-val ?p ?v ?l)
    )
  :formula (ord n1 n2)
)

(declare-method unload-top (?p ?v ?l)
   expansion(
    (n1 unload-haz ?p ?v ?l)
    (n2 unload-val ?p ?v ?l)
    )
  :formula (ord n2 n1)
)

(declare-method load-haz (?p ?v ?l)
  :expansion(
    (n warning-signs-affixed ?p ?v)
    )
  :formula (initially (type ?p hazardous))
)

(declare-method load-haz (?p ?v ?l)
  .expansion(
    (n do-nothing)
    )
  ·formula (initially (~type ?p hazardous))
)
(declare-method unload-haz (?p ?v ?l)
  :expansion(
    (n1 decontaminated-interior ?v)
    (n2 ~warning-signs-affixed  ?v)
    )
  ·formula (and
   (initially (type ?p hazardous))
   (ord n1 n2))
)

(declare-method unload-haz (?p ?v ?l)
   expansion(
    (n do-nothing)
    )
  :formula (initially (~type ?p hazardous))
)

(declare-method load-val (?p ?v ?l)
  ·expansion(
    (n do-nothing)
    )
  :formula (initially (~type ?p valuable))
)

(declare-method unload-val (?p ?v ?l)
  ·expansion(
    (n do-nothing)
    )
  :formula (initially (~type ?p valuable))
)


(declare-method load-val (?p ?v ?l)
  ·expansion(
    (n1 guard-outside ?v)
    (n2 load ?p ?v ?l)
    (n3 guard-inside ?v)
    )
   formula (and
    (ord n1 n2)
    (ord n2 n3)
    (initially (type ?p valuable))
    )
)

(declare-method unload-val (?p ?v ?l)
  ·expansion(
    (n1 guard-outside ?v)
    (n2 unload ?p ?v ?l)
    )
  :formula (and
    (ord n1 n2)
    (initially (type ?p valuable)))
)
```

```
;;; declare-method for loading REGULAR truck or traincar
(declare-method load (?p ?v ?l)
        expansion (
                (n1  door-open ?v)
                (n2  load-package ?p ?v ?l)
                (n3  ~door-open ?v)
               )
        formula (and
                 (ord n1 n2)
     (ord n2 n3)
                 (initially (type ?v regular)  )
                 (between (at-package ?p ?l)  n1 n2)
                 (between  (at-package ?p ?v)  n2  n3)
                 (between (at-vehicle ?v ?l)  n1 n3)
                 (between  (door-open  ?v)     n1  n3)
                )
)

;;; declare-method for unloading REGULAR truck or traincar
(declare-method unload (?p ?v ?l)
        :expansion (
                (n1    door-open ?v)
                (n2    unload-package ?p ?v ?l)
                (n3    ~door-open ?v)
               )
        :formula (and
                 (ord n1 n2)
     (ord n2 n3)
                 (initially (type ?v regular)  )
                 (before (at-package ?p ?v)  n1)
                 (before (at-vehicle ?v ?l)  n1)
                 (before (at-vehicle ?v ?l)  n3)
                 (before (at-package ?p ?v)  n2)
                 (between  (door-open  ?v)     n1  n3)
                )
)

;;; ------------
;;; declare-method for loading FLATBED truck or traincar
(declare-method load (?p ?v ?l)
        :expansion (
                (n1  pick-up-package-ground   ?p ?c ?l)
                (n2  put-down-package-vehicle ?p ?c ?v ?l)
               )
        :formula (and
                 (ord n1 n2)
                 (initially (type ?v flatbed)     )
                 (initially (type ?c crane)       )
                 (before (empty ?c)           n1)
                 (before (at-package   ?p ?l)  n1)
                 (before (at-equipment ?c ?l)  n1)
                 (between  (at-package   ?p ?c)  n1  n2)
                 (before (at-vehicle   ?v ?l)  n2)
                 (between (at-equipment ?c ?l)  n1 n2)
                )
)

;;; declare-method for unloading FLATBED truck or traincar
(declare-method unload (?p ?v ?l)
        :expansion (
                (n1  pick-up-package-vehicle  ?p ?c ?v ?l)
                (n2  put-down-package-ground ?p ?c ?l)
               )
         formula (and
                  (ord n1 n2)
                  (initially (type ?v flatbed)     )
                  (before (at-package   ?p ?v)  n1)
                  (before (at-vehicle   ?v ?l)  n1)
                  (before (at-equipment ?c ?l)  n1)
                  (initially (type ?c crane)       )
                  (before (empty ?c)           n1)
                  (between  (at-package   ?p ?c)  n1  n2)
                  (between (at-equipment ?c ?l)  n1 n2)
                 )
)

;;; declare-method for loading HOPPER truck or traincar
(declare-method load (?p ?v ?l)
         expansion (
                (n1    chute-connected ?v)
                (n2    fill-hopper ?p ?v ?l)
                (n3    ~chute-connected ?v)
               )
        :formula (and
                  (ord n1 n2)
```

```
     (ord n2 n3)
                (initially (type ?v hopper)  )
                (before (at-package ?p ?l)  n1)
                (before (at-vehicle ?v ?l)  n1)
                (before (at-package ?p ?l)  n2)
                (between  (at-package ?p ?v)  n2  n3)
                (before (at-vehicle ?v ?l)    n3)
                (between  (chute-connected ?v)  n1  n3)
               )
)

;;; declare-method for unloading HOPPER truck or traincar
(declare-method unload (?p ?v ?l)
        expansion (
                (n1  chute-connected ?v)
                (n2  empty-hopper ?p ?v ?l)
                (n3  ~chute-connected ?v)
               )
       :formula (and
                (ord n1 n2)
     (ord n2 n3)
                (initially (type ?v hopper)    n1)
                (before (at-package ?p ?v)  n1)
                (before (at-vehicle ?v ?l)  n1)
                (before (at-package ?p ?v)  n2)
                (before (at-vehicle ?v ?l)  n3)
                (between  (chute-connected ?v)  n1  n3)
               )
)


;;; declare-method for loading TANKER truck or traincar
(declare-method load (?p ?v ?l)
        .expansion (
                (n1    hose-connected ?v )
                (n2    valve-open ?v)
                (n3    fill-tank ?v ?p ?l)
                (n4    ~valve-open ?v)
                (n5    ~hose-connected ?v )
               )
         :formula (and
                (ord n1 n2)
     (ord n2 n3)
          (ord n3 n4)
     (ord n4 n5)
                (initially (type ?v tanker)  )
                (before (at-package ?p ?l)  n1)
                (before (at-vehicle ?v ?l)  n1)
                (before (at-package ?p ?l)  n2)
                (between  (at-package ?p ?v)  n2  n3)
                (before (at-vehicle ?v ?l)  n3)
                (between  (hose-connected ?v )  n1  n5)
                (between  (valve-open     ?v)    n2  n4)
               )
)

;;; declare-method for unloading TANKER truck or traincar
(declare-method unload (?p ?v ?l)
        expansion (
                (n1    hose-connected ?v )
                (n2    valve-open ?v)
                (n3    empty-tank ?v ?p ?l)
                (n4    ~valve-open ?v)
                (n5    ~hose-connected ?v )
               )
       ·formula (and
                (ord n1 n2)
     (ord n2 n3)
          (ord n3 n4)
     (ord n4 n5)
                (initially (type ?v tanker)    )
                (before (at-package ?p ?v)  n1)
                (before (at-vehicle ?v ?l)  n1)
                (between  (at-package ?p ?v)  n2 n3)
                (between (at-vehicle ?v ?l)  n1 n5)
                (between  (hose-connected ?v )  n1  n5)
                (between  (valve-open     ?v)    n2  n4)
               )
)


;;; declare-method for loading LIVESTOCK
(declare-method load (?p ?v ?l)
       :expansion (
                (n1    ramp-down ?v)
                (n2    trough-full ?v)
                (n3    load-livestock ?p ?v ?l)
```

```
                (n4    ~ramp-down ?v)
                )
        .formula (and
                (ord n1 n2)
      (ord n2 n3)
            (ord n3 n4)
                (initially (type ?v livestock)  )
                (initially (type ?p livestock)  )
                (before (at-package ?p ?l)  n1)
                (before (at-vehicle ?v ?l)  n1)
                (before (at-package ?p ?l)  n2)
                (between  (at-package ?p ?v)  n3  n4)
                (before (at-vehicle ?v ?l)  n4)
                (between  (ramp-down  ?v)    n1  n3)
                (between  (trough-full ?v)    n2  n4)
                )
)

;;; declare-method for unloading LIVESTOCK
(declare-method unload (?p ?v ?l)
        :expansion (
                (n1    ramp-down ?v)
                (n2    unload-livestock ?p ?v ?l)
                (n3    clean-interior  ?v)
                (n4    ~ramp-down ?v)
                )
         formula (and
                (ord n1 n2)
      (ord n2 n3)
            (ord n3 n4)
                (initially (type ?v livestock)  )
                (initially (type ?p livestock)  )
                (before (at-package ?p ?v)  n1)
                (before (at-vehicle ?v ?l)  n1)
                (before (at-package ?p ?v)  n2)
                (before (at-vehicle ?v ?l)  n4)
                (between  (ramp-down  ?v)    n1  n3)
                )
)


;;; declare-method for loading AUTO truck or traincar with CARS
(declare-method load (?p ?v ?l)
        :expansion (
                (n1    ramp-down ?v)
                (n2    load-cars ?p ?v ?l)
                (n3    ~ramp-down ?v)
                )
         formula (and
                (ord n1 n2)
      (ord n2 n3)
                (initially (type ?v auto)   )
                (initially (type ?p cars)   )
                (before (at-package ?p ?l)  n1)
                (before (at-vehicle ?v ?l)  n1)
                (before (at-package ?p ?l)  n2)
                (between  (at-package ?p ?v)  n2  n3)
                (before (at-vehicle ?v ?l)  n3)
                (between  (ramp-down  ?v)  n1  n3)
                )
)

;;; declare-method for unloading AUTO truck or traincar with CARS
(declare-method unload (?p ?v ?l)
         expansion (
                (n1    ramp-down ?v)
                (n2    unload-cars ?p ?v ?l)
                (n3    ~ramp-down ?v)
                )
        :formula (and
                (ord n1 n2)
      (ord n2 n3)
                (initially (type ?v auto)   )
                (initially (type ?p cars)   )
                (before (at-package ?p ?v)  n1)
                (before (at-vehicle ?v ?l)  n1)
                (before (at-package ?p ?v)  n2)
                (before (at-vehicle ?v ?l)  n3)
                (between  (ramp-down  ?v )  n1  n3)
                )
)


;;; declare-method for loading AIRPLANE
(declare-method load (?p ?v ?l)
        .expansion (
                (n1    ramp-connected ?v ?r ?l)
```

```
                (n2   door-open ?v)
                (n3   load-package ?p ?v ?l)
                (n4   ~door-open ?v)
                (n5   ~ramp-connected ?v ?r ?l)
               )
        :formula (and
                 (ord n1 n2)
      (ord n2 n3)
           (ord n3 n4)
      (ord n4 n5)
                 (initially (type ?v airplane) )
                 (initially (type ?r plane-ramp) )
                 (before (ramp-available ?r)  n1)
                 (before (at-package ?p ?l)  n1)
                 (before (at-vehicle ?v ?l)  n1)
                 (before (at-equipment ?r ?l)  n1)
                 (before (at-package ?p ?l)  n3)
                 (between (at-package ?p ?v)  n3  n5)
                 (before (at-vehicle ?v ?l)  n5)
                 (between (ramp-connected ?r ?p)  n1  n5)
                 (between (door-open ?v)        n2  n4)
                )
)


;;; declare-method for unloading AIRPLANE
(declare-method unload (?p ?v ?l)
        :expansion (
                 (n1   ramp-connected ?v ?r ?l)
                 (n2   door-open ?v)
                 (n3   unload-package ?p ?v ?l)
                 (n4   ~door-open ?v)
                 (n5   ~ramp-connected ?r)
                )
        .formula (and
                 (ord n1 n2)
      (ord n2 n3)
           (ord n3 n4)
      (ord n4 n5)
                 (initially (type ?v airplane) )
                 (initially (type ?r plane-ramp) )
                 (before (ramp-available ?r)  n1)
                 (before (at-package ?p ?v)  n1)
                 (before (at-vehicle ?v ?l)  n1)
                 (before (at-equipment ?r ?l)  n1)
                 (before (at-package ?p ?v)  n3)
                 (before (at-vehicle ?v ?l)  n5)
                 (between (ramp-connected ?r ?p)  n1  n5)
                 (between (door-open ?v)        n2  n4)
                )
)

,

;


(declare-method door-open (?v)
:expansion ((n1 open-door ?v)))
(declare-method ~door-open (?v)
·expansion ((n1 close-door ?v)))

,————————————————————

(declare-method chute-connected(?v)
·expansion((n1 connect-chute ?v)))
(declare-method ~chute-connected(?v)
.expansion((n1 disconnect-chute ?v)))

;————————————————————

(declare-method hose-connected(?v)
·expansion((n1 connect-hose ?v)))
(declare-method ~hose-connected(?v)
 expansion((n1 disconnect-hose ?v)))

;————————————————————

(declare-method valve-open(?v)
:expansion ((n1 open-valve ?v)))
(declare-method ~valve-open(?v)
:expansion ((n1 close-valve ?v)))

;————————————————————

(declare-method ramp-down(?v)
.expansion((n1 lower-ramp ?v)))
```

```
(declare-method ~ramp-down(?v)
:expansion((n1 raise-ramp ?v)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(declare-method trough-full(?v)
:expansion((n fill-trough ?v)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(declare-method clean-interior(?v)
:expansion((n do-clean-interior ?v)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(declare-method ramp-connected(?v ?r ?l)
:expansion((n attach-conveyor-ramp ?v ?r ?l)))
(declare-method ~ramp-connected(?v ?r ?l)
:expansion((n detach-conveyor-ramp ?v ?r ?l)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(declare-method guard-outside(?v)
 expansion( (n post-guard-outside ?v)))
(declare-method ~guard-outside(?v)
 expansion( (n remove-guard ?v)))
(declare-method guard-inside(?v)
 expansion( (n post-guard-inside ?v)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(declare-method warning-signs-affixed  (?v)
 expansion ( (n affix-warning-signs ?v)))
(declare-method ~warning-signs-affixed  (?v)
:expansion ( (n remove-warning-signs ?v)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
```

B.4 A Sample Problem

```
;


(initially-true
(RV-COMPATIBLE AIR-ROUTE AIRPLANE)
(RV-COMPATIBLE RAIL-ROUTE TRAINCAR)
(RV-COMPATIBLE RAIL-ROUTE TRAIN)
(RV-COMPATIBLE ROAD-ROUTE TRUCK)
(PV-COMPATIBLE CARS AUTO)
(PV-COMPATIBLE LIVESTOCK LIVESTOCK)
(PV-COMPATIBLE MAIL AIRPLANE)
(PV-COMPATIBLE MAIL MAIL)
(PV-COMPATIBLE VALUABLE ARMORED)
(PV-COMPATIBLE PERISHABLE REFRIGERATED)
(PV-COMPATIBLE GRANULAR HOPPER)
(PV-COMPATIBLE LIQUID TANKER)
(PV-COMPATIBLE BULKY FLATBED)
(PV-COMPATIBLE REGULAR AIRPLANE)
(PV-COMPATIBLE REGULAR MAIL)
(PV-COMPATIBLE REGULAR FLATBED)
(PV-COMPATIBLE REGULAR REGULAR)


(TYPE REGION1 REGION)
(TYPE REGION2 REGION)
(PC-COMPATIBLE CITY1 HAZARDOUS)
(IN-REGION CITY1 REGION1)
(TYPE CITY1 CITY)
(TYPE CITY1-CL1 NOT-TCENTER)
(IN-CITY CITY1-CL1 CITY1)
(TYPE CITY1-CL1 CLOCATION)
(TYPE CITY1-CL2 NOT-TCENTER)
(IN-CITY CITY1-CL2 CITY1)
(TYPE CITY1-CL2 CLOCATION)
(SERVES CITY1-TS1 CITY1)
(TYPE CITY1-TS1 NOT-HUB)
(AVAILABLE CITY1-TS1)
(TYPE CITY1-TS1 TCENTER)
(IN-CITY CITY1-TS1 CITY1)
(TYPE CITY1-TS1 TRAIN-STATION)
```

(SERVES CITY1-TS2 CITY1)
(TYPE CITY1-TS2 NOT-HUB)
(AVAILABLE CITY1-TS2)
(TYPE CITY1-TS2 TCENTER)
(IN-CITY CITY1-TS2 CITY1)
(TYPE CITY1-TS2 TRAIN-STATION)
(SERVES CITY1-AP1 CITY1)
(TYPE CITY1-AP1 NOT-HUB)
(AVAILABLE CITY1-AP1)
(TYPE CITY1-AP1 TCENTER)
(IN-CITY CITY1-AP1 CITY1)
(TYPE CITY1-AP1 AIRPORT)
(SERVES CITY1-AP2 CITY1)
(TYPE CITY1-AP2 NOT-HUB)
(AVAILABLE CITY1-AP2)
(TYPE CITY1-AP2 TCENTER)
(IN-CITY CITY1-AP2 CITY1)
(TYPE CITY1-AP2 AIRPORT)
(PC-COMPATIBLE CITY2 HAZARDOUS)
(IN-REGION CITY2 REGION2) (TYPE CITY2 CITY)
(TYPE CITY2-CL1 NOT-TCENTER)
(IN-CITY CITY2-CL1 CITY2)
(TYPE CITY2-CL1 CLOCATION)
(SERVES CITY2-AP1 CITY2)
(TYPE CITY2-AP1 NOT-HUB)
(AVAILABLE CITY2-AP1)
(TYPE CITY2-AP1 TCENTER)
(IN-CITY CITY2-AP1 CITY2)
(TYPE CITY2-AP1 AIRPORT)
(SERVES CITY2-TS1 CITY2)
(TYPE CITY2-TS1 NOT-HUB)
(AVAILABLE CITY2-TS1)
(TYPE CITY2-TS1 TCENTER)
(IN-CITY CITY2-TS1 CITY2)
(TYPE CITY2-TS1 TRAIN-STATION)
(PC-COMPATIBLE CITY3 HAZARDOUS)
(IN-REGION CITY3 REGION1)
(TYPE CITY3 CITY)
(TYPE CITY3-CL1 NOT-TCENTER)
(IN-CITY CITY3-CL1 CITY3)
(TYPE CITY3-CL1 CLOCATION)
(SERVES CITY3-AP1 CITY3)
(TYPE CITY3-AP1 NOT-HUB)
(AVAILABLE CITY3-AP1)
(TYPE CITY3-AP1 TCENTER)
(IN-CITY CITY3-AP1 CITY3)
(TYPE CITY3-AP1 AIRPORT)
(SERVES CITY3-TS1 CITY3)
(TYPE CITY3-TS1 NOT-HUB)
(AVAILABLE CITY3-TS1)
(TYPE CITY3-TS1 TCENTER)
(IN-CITY CITY3-TS1 CITY3)
(TYPE CITY3-TS1 TRAIN-STATION)
(SERVES REGION1-AP1 REGION1)
(TYPE REGION1-AP1 HUB)
(AVAILABLE REGION1-AP1)
(TYPE REGION1-AP1 TCENTER)
(IN-CITY REGION1-AP1 CITY2)
(TYPE REGION1-AP1 AIRPORT)
(SERVES REGION1-TS1 REGION1)
(TYPE REGION1-TS1 HUB)
(AVAILABLE REGION1-TS1)
(TYPE REGION1-TS1 TCENTER)
(IN-CITY REGION1-TS1 CITY3)
(TYPE REGION1-TS1 TRAIN-STATION)
(AVAILABLE ROAD-ROUTE-1)
(CONNECTS ROAD-ROUTE-1 ROAD-ROUTE CITY3 CITY1)
(CONNECTS ROAD-ROUTE-1 ROAD-ROUTE CITY1 CITY3)
(AVAILABLE ROAD-ROUTE-2)
(CONNECTS ROAD-ROUTE-2 ROAD-ROUTE CITY3 CITY2)
(CONNECTS ROAD-ROUTE-2 ROAD-ROUTE CITY2 CITY3)
(AVAILABLE AIR-ROUTE-1)
(CONNECTS AIR-ROUTE-1 AIR-ROUTE CITY2-AP1 CITY1-AP1)
(CONNECTS AIR-ROUTE-1 AIR-ROUTE CITY1-AP1 CITY2-AP1)
(AVAILABLE AIR-ROUTE-2)
(CONNECTS AIR-ROUTE-2 AIR-ROUTE REGION1-AP1 CITY1-AP1)
(CONNECTS AIR-ROUTE-2 AIR-ROUTE CITY1-AP1 REGION1-AP1)
(AVAILABLE AIR-ROUTE-3)
(CONNECTS AIR-ROUTE-3 AIR-ROUTE REGION1-AP1 CITY3-AP1)
(CONNECTS AIR-ROUTE-3 AIR-ROUTE CITY3-AP1 REGION1-AP1)
(AVAILABLE AIR-ROUTE-4)
(CONNECTS AIR-ROUTE-4 AIR-ROUTE CITY1-AP2 CITY1-AP1)
(CONNECTS AIR-ROUTE-4 AIR-ROUTE CITY1-AP1 CITY1-AP2)
(AVAILABLE RAIL-ROUTE-1)
(CONNECTS RAIL-ROUTE-1 RAIL-ROUTE CITY1-TS2 CITY1-TS1)
(CONNECTS RAIL-ROUTE-1 RAIL-ROUTE CITY1-TS1 CITY1-TS2)
(AVAILABLE RAIL-ROUTE-2)

```
(CONNECTS RAIL-ROUTE-2 RAIL-ROUTE CITY2-TS1 CITY1-TS1)
(CONNECTS RAIL-ROUTE-2 RAIL-ROUTE CITY1-TS1 CITY2-TS1)
(AVAILABLE RAIL-ROUTE-3)
(CONNECTS RAIL-ROUTE-3 RAIL-ROUTE REGION1-TS1 CITY1-TS1)
(CONNECTS RAIL-ROUTE-3 RAIL-ROUTE CITY1-TS1 REGION1-TS1)
(AVAILABLE RAIL-ROUTE-4)
(CONNECTS RAIL-ROUTE-4 RAIL-ROUTE REGION1-TS1 CITY3-TS1)
(CONNECTS RAIL-ROUTE-4 RAIL-ROUTE CITY3-TS1 REGION1-TS1)
(RAMP-AVAILABLE RAMP1A)
(AVAILABLE RAMP1A)
(AT-EQUIPMENT RAMP1A CITY1-AP1)
(TYPE RAMP1A PLANE-RAMP)
(RAMP-AVAILABLE RAMP1B)
(AVAILABLE RAMP1B)
(AT-EQUIPMENT RAMP1B CITY1-AP2)
(TYPE RAMP1B PLANE-RAMP)
(RAMP-AVAILABLE RAMP2)
(AVAILABLE RAMP2)
(AT-EQUIPMENT RAMP2 CITY3-AP1)
(TYPE RAMP2 PLANE-RAMP)
(RAMP-AVAILABLE RAMP3)
(AVAILABLE RAMP3)
(AT-EQUIPMENT RAMP3 CITY2-AP1)
(TYPE RAMP3 PLANE-RAMP)
(RAMP-AVAILABLE RAMP4)
(AVAILABLE RAMP4)
(AT-EQUIPMENT RAMP4 REGION1-AP1)
(TYPE RAMP4 PLANE-RAMP)
(AVAILABLE ROAD-ROUTE-I1547)
(CONNECTS ROAD-ROUTE-I1547 ROAD-ROUTE CITY2 CITY1)
(AVAILABLE ROAD-ROUTE-I1548)
(CONNECTS ROAD-ROUTE-I1548 ROAD-ROUTE CITY1 CITY2)
(AT-PACKAGE PKG-1 CITY1-CL1)
,

;

(TYPE PKG-1 NOT-HAZARDOUS)
(TYPE PKG-1 VALUABLE)
(TYPE PKG-1 REGULAR)
(GUARD-INSIDE TRUCK-1)
(AVAILABLE TRUCK-1)
(TYPE TRUCK-1 NOT-TRAINCAR)
(AT-VEHICLE TRUCK-1 CITY1-CL2)
(TYPE TRUCK-1 ARMORED)
(TYPE TRUCK-1 REGULAR)
(TYPE TRUCK-1 TRUCK))

,sample input task network
(setq in-tn (create-tn T (n at-package PKG-1 CITY1-CL2)))

,
```

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1976.

[2] J. Allen, J. Hendler, and A. Tate. *Readings in Planning.* Morgan Kaufman, 1990.

[3] James Allen, Henry Kautz, Richard Pelavin, and Josh Tenenberg. *Reasoning about Plans.* Morgan-Kaufmann, 1991.

[4] Scott Andrews, Brian Kettler, Kutluhan Erol, and Jaes Hendler. UM Translog: A planning domain for the development and benchmarkingof planning systems. Technical Report CS-TR-3487, University of Maryland at College Park, Dept. of Computer Science, 1995.

[5] F. Baader. A formal definition for expressive power of knowledge representation languages. In *9th European Conference on Artificial Intelligence*, Stockholm, Sweden, August 1990.

[6] F. Bacchus and Q. Yang. Downward refinement, and the efficiency of hiearchical problem solving. *Artificial Intelligence*, Vol. 71, 1994.

[7] C. Backstrom and P. Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proc. IJCAI-95*, Montreal, 1995.

[8] Christer Backstrom and Inger Klein. Planning in polynomial time: the sas-pubs class. *Computational Intelligence*, pages pp. 181–197, 1991.

[9] A. Barrett. Frugal task decomposition. Technical Report Unpublished manuscript, Computer Science Dept., University of Washington, Seattle, Washington, June 1995.

[10] A. Barrett and D. Weld. Partial order planning. Technical Report TR 92-05-01, Computer Science Dept., University of Washington, Seattle, Washington, June 1992.

[11] J. Blythe, O. Etzioni, Y. Gil, R. Joseph, A. Perez, S. Reilly, M. Veloso, and X. Wang. Prodigy 4.0: The manual and tutorial. Technical report, School of Computer Science, Carnegie Melon University, Pittsburg, Pennsylvenia, 1992.

[12] T. Bylander. Complexity results for planning. In *IJCAI-91*, pages 274–279, 1991.

[13] T. Bylander. Complexity results for extended planning. In *First Internat. Conf. AI Planning Systems*, 1992.

[14] T. Bylander. An avarage-case analysis of planning. In *AAAI-93*, pages 480–485, 1993.

[15] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[16] Gregg Collins and Louise Pryor. Achieving the functionality of filter conditions in a partial order planner. In *AAAI-92*, pages 375—380, 1992.

[17] K. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence*, November 1991.

[18] Thomas Dean. Time map maintenance. Technical Report TR 289, Yale University, New Haven, Connecticut, October 1983.

[19] M. E. Drummond. Refining and extending the procedural net. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 667–674. Morgan Kaufman, 1990.

[20] Kutluhan Erol, James Hendler, and Dana Nau. Complexity results for hierarchical task-network planning. *Annals of Mathematics and Artificial Intelligence*, (CS-TR-3240, UMIACS-TR-94-32), 1994.

[21] Kutluhan Erol, James Hendler, and Dana Nau. Semantics for hierarchical task network planning. Technical Report CS-TR-3239, UMIACS-TR-94-31, Computer Science Dept., University of Maryland, College Park, Maryland, March 1994.

[22] Kutluhan Erol, Dana Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 1995. To appear.

[23] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 189–206. Morgan Kaufman, 1990.

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

[25] E. C. Freuder and A. K. Mackworth. Special volume on constraint-based reasoning. *Artificial Intelligence*, 58, 1992.

[26] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[27] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, 1989.

[28] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, bf 5 :2-3:223–254, 1992.

[29] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 2-3:223–254, 1992.

[30] Steven Hanks and Daniel S. Weld. Systematic adaptation for case-based planning. In *First Internat. Conf. AI Planning Systems*, pages 96–105, June 1992.

[31] James Hendler. *AI Planning Systems.* Morgan Kaufmann Publishing, Palo Alto, CA. To appear.

[32] Hopcroft and Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Company Inc., 1979.

[33] Subbara Kambhampati and James. Hendler. A validation structure based theory of plan modification and reuse. *Artificial Intelligence*, May 1992.

[34] Subbarao Kambhampati. Characterizing multi-contributor causal structures for planning. In *First Internat. Conf. AI Planning Systems*, Tempe, Arizona, June 1992.

[35] Subbarao Kambhampati. On the utility of systematicity: understanding trade-offs between redundancy and commitment in partial-ordering planning. Technical report, Arizona State University, Tempe, Arizona, December 1992.

[36] Subbarao Kambhampati. On the utility of systematicity: Understanding tradeoffs between redundancy and commitment in partial-order planning. In *IJCAI-93*, Chambery, France, 1993.

[37] Subbarao Kambhampati, Craig A. Knoblock, and Qiang Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence*, To appear. To appear.

[38] Subbarao Kambhampati and Dana S. Nau. On the nature and role of modal truth criteria in planning. *Artificial Intelligence*, 1994.

[39] Laveen Kanal and V. Kumar. *Search in Artificial Intelligence.* Springer-Verlag, 1988.

[40] Brian. P. Kettler. Case-based planning with a massively parallel memory. Technical report, Dept. of Computer Science, University of Maryland,College Park, College Park, MD., 1995. In Preparation.

[41] Craig A. Knoblock. An analysis of abstrips. In *Proceedings of the first International conference on AI Planning Systems*, pages 126–135, 1992.

[42] Janet Kolodner. *Case-Based Reasoning.* Morgan Kaufmann Publishers, 1993.

[43] Amy L. Lansky. A representation of parallel activity based on events, structure, and causality. In M. Georgeff and A. Lansky, editors, *Reasoning About Actions and Plans*, pages 123–160. Morgan Kaufmann, 1987.

[44] Amy L. Lansky. Localized event-based reasoning for multiagent domains. *Computational Intelligence Journal*, 1988.

[45] Vladimir Lifschitz. On the semantics of strips. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 523–531. Morgan Kaufman, 1990.

[46] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *AAAI-91*, 1991.

[47] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, bf 4:463–502, 1969.

[48] D. McDermott. Flexibility and efficiency in a computer program for designing circuits. Technical Report TR-40, Massachusetts Institute of Technology, 1977.

[49] S. Minton, J. Bresna, and M. Drummond. Commitment strategies in planning. In *IJCAI-91*, 1991.

[50] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, December 1992.

[51] A. Newell and J. A. Simon. Gps, a program that simulates human thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. R. Oldenbourg KG, 1963, 1963.

[52] N. Nilsson. *Principles of Artificial Intelligence.* Morgan-Kaufmann, 1980.

[53] Edwin P. D. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, bf 4:356–372, 1988.

[54] J.S. Penberthy and D.S. Weld. Ucpop: A sound, complete, partial order planner for adl. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, October 1992.

[55] M. A. Peot. Conditional nonlinear planning. In *First Internat. Conf. AI Planning Systems*, pages 189–197, 1992.

[56] M. A. Peot and D. Smith. Threat removal strategies for partial-order planning. In *AAAI-93*, pages 492–499, Washington D.C., 1993.

[57] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 1995.

[58] E. D. Sacerdoti. *A Structure for Plans and Behaviour.* Elsevier-North Holland, 1977.

[59] E. D. Sacerdoti. The nonlinear nature of plans. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 162–170. Morgan Kaufman, 1990.

[60] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 98–109. Morgan Kaufman, 1990.

[61] D. Smit and M. A. Peot. Postponing threads in partial-order planning. In *AAAI-93*, pages 500–506, Washington D.C., 1993.

[62] Mark Stefik. lanning with constriants. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 171–185. Morgan Kaufman, 1990.

[63] G.J. Sussman. *A Computational Model of Skill Acquisition.* American Elsevie, 1975.

[64] A. Tate. Generating project networks. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 291—296. Morgan Kaufman, 1990.

[65] A. Tate, B. Drabble, and J. Dalton. The use of condition types to restrict search in an ai planner. In *AAAI-94*, pages 1129–1134, 1994.

[66] A. Tate, J. Hendler, and D. Drummond. Ai planning: Systems and techniques. *AI Magazine*, UMIACS-TR-90-21, CS-TR-2408:61–77, 1990.

[67] Manuela Veloso. Learning by analogical reasoning in general problem solving. Technical report, Carnegie Mellon University, School of Computer Science, 1992.

[68] Manuela Veloso and Jim Blythe. Linkability: Examining causal link commitments in partial-order planning. In *Second Internat. Conf. AI Planning Systems*, Chicago, 1994.

[69] S. A. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):246–247, 1983.

[70] R. Waldinger. Achieving sevaral goals simultaneously. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 118—139. Morgan Kaufman, 1990.

[71] D. H. D. Warren. Extract for apic studies in data processing. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 140–153. Morgan Kaufman, 1990.

[72] D. Wilkins. *Practical Planning: Extending the classical AI planning paradigm.* Morgan-Kaufmann, 1988.

[73] D. E. Wilkins. Domain-independent planning: Representation and plan generation. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 319–335. Morgan Kaufman, 1990.

[74] Q. Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6:12–2, 1990.

[75] Q. Yang. Understanding the essence of nonlinear planning. Technical report, Computer Science Department, University of Waterloo, 1991.

[76] Q. Yang, D. S. Nau, and J. Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, February 1993.

[77] Q. Yang, J. Tenenberg, and S. Woods. Abstraction in nonlinear planning. Technical Report TR CS-91-65, University of Waterloo, Dept. of Computer Science, 1991.

[78] R. M. Young, M. E. Pollack, and J. D. Moore. Decomposition and causality in partial-order planning. In *Second Internat. Conf. AI Planning Systems*, Chicago, 1994.