Abstract

Title of Dissertation: Case-based Planning with a
High-Performance Parallel Memory

Brian Patrick Kettler, Doctor of Philosophy, 1995

Dissertation directed by: Associate Professor James Hendler
Department of Computer Science

Case-based planning (CBP) systems, like other case-based reasoning systems, can take advantage of previous planning experience by reusing stored cases (plans) in similar situations in the future. Advantages of CBP include speedup over planning from scratch and the ability to function with limited causal domain knowledge. "Traditional" CBP systems with the latter advantage typically cannot produce plans from scratch because they lack the more powerful adaptation mechanisms of generative planning systems. These "reuse-only" CBP systems rely on retrieving a plan from the casebase that is close to a solution plan. This requires large casebases with good coverage of the problem space and the ability to encode and match cases at fine levels of detail.

Many such CBP systems, however, have fallen short of these requirements. They support only small, pre-indexed casebases. Pre-indexing constrains retrieval, as does the use of less expressive feature-based case representation schemes. The encoding and matching of detailed structural relationships in cases is not possible in such systems. These systems often adapt a single plan to the target problem using methods that are ad hoc or heuristic.

CAPER is a novel, domain-independent, case-based planning system with improvements over traditional reuse-only CBP systems from its use of techniques that exploit a high-performance parallel memory of cases. CAPER takes a memory-intensive approach by making frequent use of memory during all phases of planning and by using large casebases, which can be automatically seeded. Because the parallel retrieval mechanisms scale to real-world sized casebases of thousands of plans, memory does not have to be pre-indexed and

thus retrieval is more flexible. Detailed queries can be used to match cases, which are stored using an expressive, graph-structured case representation scheme.

Plan adaptation in CAPER borrows techniques from generative planning, such as the use of plan validations, which capture dependencies in a plan, and plan composition. These techniques are incorporated into a reuse-only CBP framework for a more principled approach to adaptation than in many reuse-only CBP systems. CAPER can also use its flexible retrieval mechanisms and case representations to retrieve patch or substitute plans from memory.

Case-based Planning with a High-Performance Parallel Memory

by

Brian Patrick Kettler

Dissertation submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1995

Advisory Committee:

Associate Professor James Hendler, Chairman/Advisor
Professor Lindley Darden
Professor Dana Nau
Assistant Professor Bonnie Dorr
Assistant Professor Lee Spector

Table of Contents

List of Tables

List of Figures

Case-based Planning with a
High-Performance Parallel Memory

Brian Patrick Kettler

November 21, 1995

This comment page is not part of the dissertation.

Typeset by LaTeX using the `dissertation` class by Pablo A. Straub, University of Maryland.

Chapter 1

Introduction

Artificial intelligence systems require much knowledge to understand the world and solve problems in it. This knowledge can be built into to an A.I. system by humans or learned by the system over the course of its operation. The former approach is useful when a large body of knowledge exists and can be encoded into a form that the A.I. system can use. The latter approach is useful when the available knowledge is impoverished, and thus the A.I. system must gather knowledge on its own. It will have to understand a new situation or solve a new problem in terms of its current knowledge. A new experience can be remembered as a *case*, an encoded description of the situation/problem and its interpretation/solution. When a similar situation is encountered in the future, the system can apply knowledge from its *casebase*, a persistent collection of cases. Reasoning using cases is called *case-based reasoning* (CBR) and has been investigated by A.I. researchers over the past fifteen years (Kolodner, 1993).

CBR is a kind of analogical reasoning. In *analogical reasoning*, knowledge from one problem or situation is applied to understand a problem or situation via an analogy. An *analogy* is a comparison between two *analogues* — two things, two models (e.g., an atom and a solar system), two events, (e.g., the Vietnam and Korean wars), or two problems (e.g., preparing a beef and broccoli stir-fry dish and a chicken and snow peas stir-fry dish). Often one analogue is a novel problem or situation that the reasoner is trying to solve or understand in terms of a more familiar problem (or situation) recalled by the reasoner. The novel problem/situation is termed the *target* analogue, and the familiar problem/situation is termed the *source* analogue. In CBR, the analogues are cases.

Cognitive scientists have long studied analogical reasoning in humans and have even constructed some computational models (e.g., (Holyoak and Thagard, 1995; Falkenhainer *et al.*, 1990; Falkenhainer *et al.*, 1990)). As will be discussed in Section 8.3.2, this work provides a framework for conceptualizing the CBR process. The technique of CBR itself was inspired by psychological studies of human memory (see Section 8.3.1). The synergy between cognitive science and CBR continues in the work described in this thesis, the CAPER system. Though CAPER is not a model of cognition, several aspects of its design were inspired by work in cognitive science, as will be discussed in Sections 4.6.5 and 9.1.3.

CBR systems have been built for many different tasks and domains including several successful commercial "expert" systems (Kolodner, 1993). It has been well-established that human experts reason using cases in a variety of domains such as law or medicine (e.g.,(Boshuizen and Schmidt, 1992)). These fields have huge repositories of knowledge in the form of cases, and new cases are interpreted in terms of old ones. A casebase can serve as a repository of expertise in an organization — a kind of institutional collective memory which does not fade or retire.[1] Novices can access this knowledge during training or when no expert is available. A person cannot possibly remember every case in a casebase of thousands of cases, particularly when there are no obvious similarities between the cases. Although they solve problems without human intervention, CBR systems can also complement the recall limitations of people by presenting relevant cases to support their decision making.

CBR systems are often easier to build than rule-based systems (Kolodner, 1993)[p. 94]. Human experts may be better able to articulate their expertise in terms of specific cases, rather than in terms of general rules. If a system is to explain its reasoning to a person, a list of cases used by the system may be more informative than a list of rules used or statistical calculations performed. Cases may also be used to complement rules in a system (e.g., (Rissland and Skalak, 1991; Kettler and Darden, 1993; Sanders, 1994)).

1.1 Planning

Many real-world problems involve planning of some kind. Planning problems range in scope from planning the day's errands to planning for military force deployment. Over the past thirty years, research in A.I. planning has focused on the development of general-purpose planning techniques applicable to *domains* such as manufacturing, transport logistics, robot navigation, etc. A *planning problem* consists of an initial world state and the set of goals to achieve. The purpose of planning is to generate a plan, a sequence of actions that achieves a set of goals when executed.[2]

A.I. planning systems have used many techniques (Allen *et al.*, 1990). This dissertation focuses on the technique of *case-based planning* (CBP), a kind of case-based reasoning first used in the CHEF system (Hammond, 1986)). A CBP solves a new *target* problem by retrieving a case — a previously solved problem and its plan — from the casebase and adapting the *source* plan to solve the target problem, possibly with the aid of feedback from plan execution. The newly generated plan is then added to the casebase for later reuse.

A.I. researchers have focused on *domain-independent* techniques, those designed to be applicable for multiple planning domains. A planning system embodying these techniques is configured for a particular domain by supplying it with *domain knowledge* in some encoded form. The assumption is that reconfiguration of the planner requires less effort than building

---

[1] The term "memory" will be used in this document to refer to either a knowledge base (KB) of a computer program or to human memory. It will *not* be used to mean physical computer memory (RAM, disk, etc.).

[2] This conception of planning as a sequence of actions that transforms an initial state into a goal state is the "classical" one. More recent research on reactive AI planning systems in dynamic environments has suggested alternate conceptions of planning.

a domain-dependent planner for every new domain.

Domain knowledge is of two basic kinds: causal knowledge and control knowledge. *Domain causal knowledge* (also known as the *domain theory*) is knowledge about the possible states of the world and the available actions. This includes knowledge about each action's *preconditions* — the states must hold before an action can be executed — and effects — the changes to the states that hold *after* an action is executed. *Domain control knowledge* includes heuristics and other information used to guide the search for a plan during planning. Domain knowledge is typically supplied by a human *domain expert* or *domain engineer*.

1.1.1 The Plan Reuse Spectrum

Planning systems can be classified according to the degree that they use previously generated plans when solving a new planning problem. Systems can be located on the spectrum shown in Figure 1.1. These distinctions between segments of the spectrum are from an engineering perspective, i.e., what techniques are used, what knowledge is required, etc. The distinctions at theoretical level — what the fundamental differences are between the various approaches are — are still being investigated (e.g., (Erol, 1995; Koehler, 1994; Hanks and Weld, 1995)).

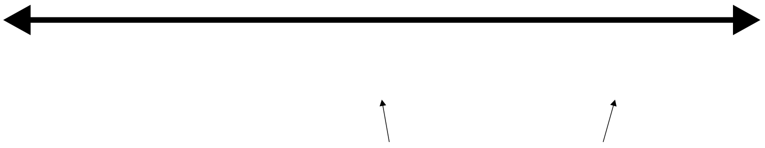

Figure 1.1: The Plan Reuse Spectrum of Planning Systems

Generative-Only Planning Systems

On one end of the spectrum are *generative-only* planning systems. They do *not* reuse previous plans but instead generate a new plan from scratch to solve a target planning problem. At an *abstract level*, planning systems can be conceived of as searching a space of plans, one or more of which solve the target problem. Planners explore this plan space by transforming a plan. Generative-only planning systems always start from the same point in the plan space — the null plan — and attempt to "find" a plan that solves the target problem by generating one from scratch. To find a solution plan, sufficiently powerful plan adaptation mechanisms are needed to transform the null plan into a solution plan.

A variety of *generative* techniques are used to produce plans, most of which involve search of some kind. Some early generative-planning systems use theorem proving and search a space of sets of clauses (e.g., (Green, 1969)). Systems like STRIPS (Fikes and Nilsson, 1971) search a space of world states. Systems like NONLIN (Tate, 1977) and UMCP (Erol, 1995) search a space of partially-generated plans. The planner must find a plan in this space that contains no harmful interactions among plan actions. The search space may be very large, even for relatively small domains. The UM TRANSLOG domain (Appendix A.1) has a search space of over ten billion task hierarchical networks.[3]

These planning systems are often called "classical" planning systems and are often designed to be sound and complete. A *sound* planner produces only correct plans with respect to its domain knowledge. A *complete* planner is able to produce a plan for each problem in the domain. For a planning system to be sound and complete, however, requires complete and correct causal domain knowledge in the form of a fully-specified set of plan operators where the preconditions and effects of all actions are all known in advance. Such strong domain causal knowledge is only available in well-understood domains (e.g., the artificial blocks world domain, some kinds of manufacturing, etc.).

Classical planning systems use search with projection in domains where one can make *static-world assumptions*: the world changes only as a result of the agent who is executing actions in the plan (i.e., no other agent is changing the world), the changes to the world are completely predictable, complete and correct knowledge of the world state is available, and the time taken for plan generation/execution is not crucial. The focus is on plan generation, not plan execution. More recent research has been on planning in more dynamic domains where these assumptions do not hold (e.g., (Agre and Chapman, 1987; Spector, 1992)).

Reuse-Optional Planning Systems

Reuse-optional planning systems can generate a plan from scratch *and* optionally reuse previously generated plans. These include systems such as PRIAR (Kambhampati and Hendler, 1992), SPA (Hanks and Weld, 1995), and PRODIGY/ANALOGY (Veloso, 1992). These systems all embody a generative planner — NONLIN, SNLP, and PRODIGY, respectively — that can solve planning problems from scratch. To the extent that their planner components are sound and complete, reuse-optional systems can transform a null plan into a solution plan for the target problem. They require the same sort of strong domain causal knowledge that generative-only systems do.

Reuse-optional systems can also take advantage of previous plans stored as "cases" and thus can be considered *case-based* planners. They can retrieve a source plan and transform it into a target plan. Given a correct and complete generative planner component, these systems can transform any source plan into any target plan.

The benefit of starting from a source plan, rather than null plan, is a potential savings in planning time because the search in the "plan space" is reduced. SPA, PRIAR, and PRODIGY/ANALOGY have demonstrated speedups ranging from 18% to 360% over planning from scratch (Hanks and Weld, 1995; Kambhampati and Hendler, 1992; Veloso,

---

[3] As estimated by Kutluhan Erol for the UM TRANSLOG domain as represented for the UMCP planner.

1992).[4] Starting from a source plan, however, may be worse than starting from scratch. The plan produced starting from a source that is highly dissimilar to the target problem may be longer than a plan produced from scratch. It may even be impossible to produce a correct plan from a misleading source plan.[5] Using an inappropriate analogue is one of the dangers of reasoning by analogy (see (Holyoak and Thagard, 1995) for some examples).

1.1.2 Reuse-Only Planning Systems

Reuse-only planning systems that cannot solve a planning problem from scratch and thus depend on plan reuse. These *case-based* planning systems include "traditional" CBP systems such as CHEF (Hammond, 1986), COOKIE (McCartney, 1993), and PLEXUS (Alterman, 1988). Reuse-only systems cannot solve a new problem from scratch because they lack the more powerful adaptation mechanisms that generative planning systems have and thus cannot transform a null plan into a target plan. Many reuse-only systems, however, do *not* require the kind of strong domain causal knowledge needed by more powerful adaptation mechanisms. Thus these systems can function in domains where strong causal knowledge is not available or where the environment is dynamic. This benefit of reuse-only systems has been the design focus of many CBP systems, rather than the potential benefit of speedup over systems that plan from scratch.

In the absence of a strong domain theory, implicit domain knowledge from a source case can be transferred to the target case. Case-base planning has been used in several domains where a strong domain theory is unavailable including law (e.g., CHIRON (Sanders, 1994)), scientific reasoning (e.g., SEQER (Kettler and Darden, 1993)), and cooking (e.g., COOKIE (McCartney, 1993)). CHIRON, for example, can plan tax strategies where open-textured concepts like "principal residence" are defined by their application in previous legal cases, rather than by sets of necessary and sufficient conditions.

Reuse-only systems usually provide no guarantees of correctness and completeness for several reasons. Given a target problem, the system may not be able to find a relevant source plan in its casebase to adapt. This may be a result of inappropriate pre-indexing of the casebase (see Section 2.3.3), faulty retrieval heuristics, or simply a lack of relevant cases in the casebase. If a strong domain theory is lacking, the system may not be able to adapt a retrieved relevant plan to the new situation. Or, such adaptation may rely on heuristic methods which are not guaranteed to produce a correct plan.

Reuse-only systems typically require feedback from plan execution to find and fix faults in the plan. This feedback can either be from a human executing the plan (e.g., COOKIE), a robot, or a simulator (e.g., CHEF). Some systems can store plan failure information in the casebase to avoid the same failure in the future (e.g., CHEF).

---

[4] In such systems, the speedup from plan reuse is relatively easy to determine empirically as plan reuse can be turned on and off.

[5] Systems like PRIAR do not suffer from this in simple domains such as blocks stacking where objects are untyped, and the planner can transform any source plan into any target plan.

Without strong domain theories and powerful adaptation mechanisms, a reuse-only CBP system will be successful to the extent that it can find a starting point (i.e., a source plan) that is "close" to a solution point (i.e., a plan for the target problem). Thus the focus in these systems is on retrieving a source plan that solves a problem similar to the target problem. This requires large casebases with good coverage of the problem-space and effective retrieval mechanisms. Retrieval methods must be scalable so that as the casebase grows in size, the retrieval time stays reasonable, especially if a speedup over planning from scratch is a design goal. The requirements for CBP systems in general, and reuse-only CBP systems in particular, are the focus of Chapter 2.

1.2 CAPER

As discussed in the previous section, *case-based planning* (CBP) systems include both reuse-optional planning systems such as PRIAR and reuse-only planning systems such as CHEF. The former can solve problems from scratch but gain a speedup from plan reuse. The latter can potentially provide a speedup benefit, but their greatest benefit is their ability to function without the kind of strong domain causal knowledge used by the more powerful adaptation mechanisms of generative planners.

Most CBP systems add plans they produce for new planning problems to the casebase. Thus a problem previously solved does not have to be solved again: its solution can just be retrieved from the casebase. This memorization of plans is *rote* learning, where the benefit is speedup. CBR (CBP) can also involve *exemplar-based learning*, where the *exemplars* are the individual cases (plans). Exemplars can be used in multiple ways. In *instance-based* learning, an exemplar (an *instance*) can be used "on demand": when a new problem is to be solved, a previous plan can be retrieved as the basis for the new plan. This is a kind of *lazy learning*, as a stored instance is not "learned from" until some similar problem comes along (Aha *et al.*, 1991).

1.2.1 Contributions

CAPER, a reuse-only, case-based planning system designed and developed by the author, is the subject of this dissertation. Because reuse-only CBP systems cannot generate a plan from scratch, they require effective retrieval to find a source plan that solves a problem similar to the target problem. This in turn requires a large casebase with diverse problem coverage and retrieval mechanisms that scale well to it. Effective adaptation mechanisms are also required to transform a source plan into a plan for the target problem.

Traditional reuse-only CBP systems have often fallen short of these requirements. Chapter 2 surveys some of these systems and their shortcomings in detail. In brief, these systems typically use serial retrieval mechanisms and thus require their relatively small casebases to be indexed for efficiency. This practice, in conjunction with simple feature-based representations that do not support the encoding of richly detailed cases, results in inflexible case retrieval. Memory is often accessed infrequently, and a single source plan is adapted to the target plan using methods that are often ad hoc.

The following is a summary of CAPER's contributions towards overcoming the limitations found in many traditional reuse-only CBP systems:

- CAPER takes a more memory-intensive CBP approach to maximize the use of a valuable knowledge commodity, the casebase.
    - Memory is accessed frequently during all phases of planning.
    - Arbitrarily large casebases can be automatically seeded using a generative planner.
- CAPER has more flexible case retrieval. Thus it is able to find source cases that more closely match the target problem and hence require less adaptation effort.
    - Memory is not pre-indexed. Retrieval probes (memory queries) can contain any feature of the target problem, and cases (plans) can be retrieved via any of their constituent features. This provides the ability to match cases at varying levels of detail.
    - Failing retrieval probes can be automatically modified by a search planning mechanism.
    - Details of cases can be captured in an expressive graph-structured case representation. Structural relationships in a case's problem description and plan can be stored and matched on later.
    - Components of plans (e.g.,subplans) can be retrieved independently.
- CAPER takes a more principled approach to plan adaptation:
    - Plan validation structures based on those of PRIAR (Kambhampati and Hendler, 1992) are adapted for mapping and adaptation in a reuse-only framework. The techniques used are domain-independent.
    - Knowledge from multiple cases can be applied to solve a single target problem. Patch or substitute plans can be recursively retrieved from memory and merged into a single plan to solve the target problem.

CAPER is designed to exploit the fast, scalable, parallel retrieval mechanisms provided by PARKA, a frame-based knowledge representation system (Evett *et al.*, 1994; Kettler *et al.*, 1995). The high-performance retrieval mechanisms of PARKA provide fast matching of retrieval probe graphs to structures in cases, even for very large casebases containing hundreds of thousands of frames.

Although its design was motivated by the availability of these high-performance retrieval mechanisms, many of CAPER's techniques can improve planning when only serial retrieval mechanisms are available. These include the use of graph-structured representations, automatic modification of retrieval probes, (recursive) retrieval and merging of component plans to solve a single problem, use of plan validations for mapping and adaptation, and automatic seeding of casebases. Given a fast parallel memory, many traditional reuse-only CBP systems memory would still only retrieve a single plan to adapt, access memory infrequently, use an ad-hoc approach to plan adaptation, etc.

The CAPER system integrates domain-independent mechanisms for plan retrieval, plan elaboration (mapping and instantiation), plan adaptation, and plan storage. The latter stores plans that CAPER produces as new cases for future reuse. CAPER's persistent knowledge is stored in a single semantic network implemented via the PARKA system.

When available, domain-specific knowledge and heuristics can be represented and taken advantage of. Though CAPER's techniques have the greatest potential for rich, real-world domains that require expressive case representations and flexible retrieval, the above techniques have been illustrated in a benchmark domain, UM TRANSLOG (Andrews *et al.*, 1995). Components of CAPER have also been used for derivational analogy in a protein sequencing experiment planning domain (Kettler and Darden, 1993) (see Section 8.2).

1.2.2 The CAPER System

CAPER is implemented in Common Lisp.[6] CAPER's memory (knowledge base) is implemented using a frame-based, parallel knowledge representation system, PARKA (see Section 3.1).

The architecture of CAPER is shown in Figure 1.2. CAPER executes the following top-level process when given a new target planning problem (goals and initial situation). Subsequent chapters describe CAPER's individual components.

1. The **Problem Analyzer** (Section 4.4) takes a planning problem specified via the Problem Specifier (Section 3.5.2) and creates a target problem description and some initial frames to represent the new case.

2. The **Plan Retriever** (Section 4.5) takes a target problem description and a retrieval plan, executes the retrieval plan until candidate source plans are found by the Retriever (Section 4.6) and PARKA Structure Matcher (Section 3.1.4), orders the candidate plans using a simple similarity metric, and returns the high-scoring source plan.

3. The **Plan Mapper** (Section 5.3) takes a chosen source plan and an initial mapping generated in the retrieval matching process and returns a mapping between objects in the source and target problems. The Plan Mapper also calls the PARKA Structure Matcher.

4. The **Plan Instantiator** (Section 5.4) takes a source plan, a target problem description, and a mapping and instantiates a new target plan.

---

[6] The prototype version of CAPER is implemented in Macintosh Common Lisp (CLOS).

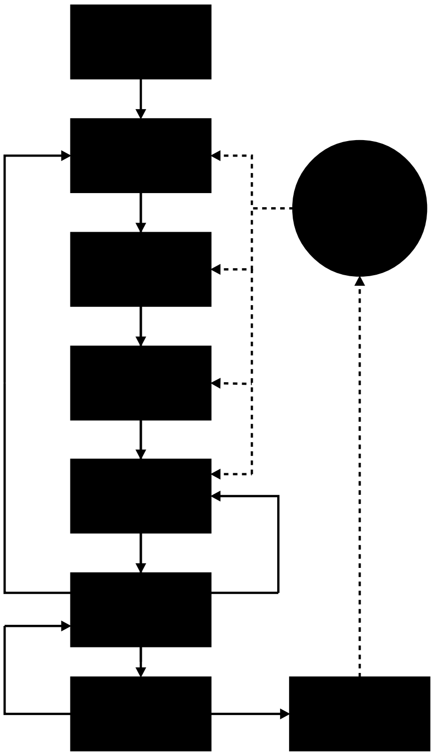

Figure 1.2: CAPER's Top-Level Architecture. (Solid arrows indicate the major flow of control. Dashed arrows indicate the flow of data.)

5. The **Plan Checker** (Section 6.1) takes a new target plan, combines it with previous plans via the Plan Merger (Section 6.3). It returns the combined plan and any predicted failures due to differences between the two problems or interactions between multiple target plans.

6. The **Plan Modifier** (Section 6.2) takes predicted failures of the target plan and adapts the target plan to remedy them. The Plan Modifier may call the Plan Retriever to retrieve a patch plan or alternate plan. It can also call the Plan Checker to evaluate changes.

7. The **Plan Tester** presents a plan to a user for evaluation/execution and returns descriptions of any failures found. Failures are then repaired by the Plan Modifier, and execution feedback stored in the casebase.[7]

8. The **Plan Storer** (Section 6.5) stores a target problem and its plan as a new case in the casebase.

---

[7] The Plan Tester component is not currently implemented. See Section 9.2.

1.3 A Guide to What Follows

Chapter 2 discusses the requirements for a system to do useful case-based planning, how they have been addressed by some relevant CBP systems, and how they have motivated CAPER's design.

Chapter 3 describes CAPER's Memory including case representation and the underlying knowledge representation system, PARKA.

Chapters 4, 5, and 6 describe retrieval, elaboration (plan mapping and instantiation), and adaptation (plan checking, merging, modification, and storage) in CAPER, respectively.

Chapter 7 presents some extended examples of CAPER's operation. Chapter 8 discusses the other connections to the CAPER research including derivational analogy and cognitive science (human memory and analogical reasoning). Chapter 9 presents some directions for future work and conclusions.

The appendices present details about CAPER's test domains (Appendix A), data structures (Appendix B), and examples (Appendix C).

Chapter 2

Motivations and Related Work

CAPER is a case-based planning (CBP) system that addresses some of the limitations of "traditional", reuse-only CBP systems. As discussed in Section 1.1.1, CBP systems have several major requirements. This chapter discusses these requirements in detail, focusing on the limitations of some previous approaches taken in other CBP systems. How these limitations have motivated the approach taken in CAPER will be discussed briefly. The details of the components of CAPER follow in subsequent chapters. Section 8.3.2 presents a conception of the analogical reasoning process that I borrow as an organizing framework for the discussion of these components.

A case-based planning system requires at least the following to produce useful plans:

- a large library of useful cases (plans);
- an expressive scheme for representing cases;
- an effective and efficient mechanism for retrieving relevant cases; and
- an effective and efficient mechanism for adapting plans.

The following sections discuss these interrelated requirements.

2.1 Large Case Libraries

This section discusses the first major requirement for CBP systems: having a large casebase. CBP systems require a casebase to reuse plans. As discussed in Section 1.1.1, reuse-optional CBP systems need cases to derive a speedup over planning from scratch. Reuse-only CBP systems need cases to produce a plan at all.

A casebase should contain good coverage of the space of possible problems, i.e., a representative set of problems and their solutions. This is desirable in reuse-optional systems because greater speedup will be derived by starting from a plan that is close to the target plan. This is desirable in reuse-only systems because they often lack adaptation mechanisms of sufficient power or the requisite strong causal domain knowledge to transform any source plan into any target plan. Such systems need to start with a plan close to the target plan to produce a plan at all. Thus the more and various source plans in the casebase, the better.

Good problem coverage for large problem spaces requires many cases and hence a large casebase. In the extreme case, the casebase has total coverage of the problem domain (i.e., it contains a case for each possible problem) and solving a new problem becomes a matter of retrieving the solution for an exactly matching previously-solved problem, with no adaptation required. While total coverage is generally not possible in any but the smallest of domains, a larger casebase can reduce and simplify adaptation.

To show "how a bigger case library can allow the use of significantly weaker adaptation rules and still get strong results", Riesbeck and Schank give the example of calculating a logarithm (Riesbeck and Schank, 1989, p. 43). Rather than calculate a logarithm from scratch, if the logs for two numbers close to the target number can be obtained from a log table, then the simpler and faster process of interpolation can be used to calculate the log of the target number. This is possible, however, only if the logs for two numbers close to the target number can be found in the log table (the "casebase"). Because it requires less domain (i.e., mathematical) knowledge, interpolation is a "weaker" adaptation "rule" than calculation from scratch.[1]

In general, bigger planning domains (i.e., greater numbers of objects, potential actions, etc.), require larger casebases to provide good coverage. For example, a transport logistics problem domain (where packages are delivered using vehicles) with 100 vehicles, 100 packages, and 100 locations has $100^{100} \times 100^{100} = 10^{400}$ possible initial situations and 100^{100} possible goals for a total of 10^{600} possible problems in the domain! As a point of comparison, United Parcel Service has 500 aircraft, 130,000 vehicles, and 2,400 facilities, 3 million daily regular customers, and delivers 11.5 million parcels and documents per day.[2]

Real-world domains can require casebases of thousands of cases or more (e.g., (Miyashita and Sycara, 1993; Kahn, 1994; Kitano *et al.*, 1992)). Unlike cases for toy domains such as block stacking ("Blocks World"), real-world cases may be much larger in size as they may each contain many facts, large plans, etc. As an example, the United States Transportation Command's library of Time-Phased Force Deployment Data (TPFDD) consists of operational plans (OPLANs), force modules (FMs), and units (ULNs) and is used for planning large-scale military force deployments. Plans in this domain can have hundreds of thousands of distinct force requirements. One OPLAN I encoded contained nearly 270 FMs and 6000 ULNs which came to over 7500 frames and 267,000 assertions in a PARKA network (Section 3.1).[3] A medical database consisting of 20,000 records of pregnancy-related doctor visits has been encoded into PARKA to support data mining applications is over 21,000 frames and a million assertions.

To support large (and growing) casebases a CBP system needs scalable retrieval tech-

---

[1] Of course domain knowledge had to be applied by someone to create the log table in the first place.

[2] These numbers are from the UPS WWW pages (as of 7/95)
(URL http://www.ups.com/about/glance.html).

[3] Assertions correspond to frame slots.

Cases	**Plans & Subplans**	**Frames**	**Assertions**
20	643	10246	129888
40	1428	21973	308225
60	2260	34148	500327
80	3077	45802	679803
100	4071	58344	899132
120	4907	69720	1071193
140	5667	82499	1295920
160	6556	94551	1500080
180	7360	105806	1665645
200	8130	116728	1746060

Table 2.1: UM TRANSLOG casebase sizes. (Frames and assertions are the logical data structures of PARKA, used to implement CAPER's KB.)

Domain	**Frames/Case**
Car Construction	85
CMU Translog	168
UM Translog	572

Table 2.2: Case sizes for several CAPER domains.

niques and automated methods for case encoding. The former are needed to keep retrieval times low, particularly in CBP systems that attempt to derive a speedup over planning from scratch. The latter are needed to reduce the effort of constructing the initial casebase.

Many research CBP systems have small casebases (e.g., (Hammond, 1986; Kolodner, 1989)). CAPER can support very large casebases. Table 2.1 shows casebase sizes for CAPER's UM TRANSLOG domain. The number of plans and subplans is the more important number, because CAPER can retrieve them individually. In other words, a case has one plan that is composed of many subplans. The average size of a case in CAPER has grown with the use of more complex domains. This is shown in Table 2.2.

Many CBP systems require hand-encoding of cases (e.g., (Alterman, 1988; Sanders, 1994)). CAPER's Casebase Seeder, described in Section 3.5, can generate an arbitrarily large initial casebase automatically. A generative planning system, UM NONLIN, is used to produce correct plans on randomly-generated problems in the UM TRANSLOG domain. These plans are then automatically encoded into CAPER's semantic net representation. As memory is not pre-indexed, no hand-indexing of cases is required (see Section 2.3.2). Thus the case acquisition process is entirely automated. CAPER also adds plans it generates to its casebase (Section 6.5).

2.2 Expressive Case Representation Scheme

In addition to having lots of cases for good problem-space coverage, a CBP system needs cases of appropriate content. Because it is desirable that a CBP system retrieve a plan close to the target problem, it must be able to match the target problem to the source case at a fine level of detail. The level of detail of a case that can be represented is a function of the case representation scheme, the topic of this section, and the retrieval/matching process, the topic of the next section.

An expressive case representation is thus a second major requirement for CBP systems. The case representation scheme specifies how the facts of a case are to be encoded (and possibly how cases are to be organized in the casebase). A casebase must support the functions of case storage, matching (retrieval and mapping), and adaptation. The first of these operations is the focus of this section. The latter two are the focus of Section 2.3 and Section 2.4. In CAPER, a graph-structured representation is used for cases in a casebase that is not pre-indexed. This representation is discussed in Section 2.2.1.

2.2.1 Graph-Structured Case Representations

Some early case representation schemes, inspired by the psychology of human memory (see Section 8.3.1). Other case representation schemes, including many feature-based schemes, were inspired by engineering concerns such as supporting efficient case acquisition, matching, etc. This section first discusses the limitations of feature-based case representation schemes and then describes an alternative — graph-structured representations — used by CAPER and several other CBR systems.

A feature-based scheme represents a case as a group of features. Simple feature-based schemes include fixed-length feature vectors (e.g., MBRTALK (Stanfill and Waltz, 1986)), variable-length feature vectors/sets (e.g., information retrieval systems where a document is represented as a set of individual words it contains), and frames with atomic slot-fillers.

A feature vector is typically a fixed length list of attribute values. For example, consider a case involving a legal dispute between two companies, Alpha Corporation, located in Kingston, New York, and Beta Corporation, located in Johnstown, Pennsylvania. A feature vector could be used to represent this case by encoding the values of each company's name and location attributes as $\langle$Alpha, NY, Beta, Penn$\rangle$.

This case could also be represented as a frame with atomic slot fillers:

```
Case-1
    company-1: Alpha
    company-2: Beta
   location-1: Kingston, New York
   location-2: Johnstown, Pennsylvania
```

These feature-based representations are not easily extendable, e.g., if one case had three companies involved, we would need a slot or vector location to be allocated in all case frames or vectors. Feature-based representations such as these do not permit arbitrary relations between attributes to be expressed, e.g., the potentially relevant fact that Alpha and Beta are located in states, New York and Pennsylvania, that border each other.

A more expressive frame representation is used by systems such as HYPO (Ashley, 1990). Slots can be filled with frames, in addition to atomic values, and thus frames can be nested. Even in the more expressive scheme, all of the potential slots typically must be specified in advance. This requirement is undesirable in domains where a case can contain many diverse facts whose relevance to the case's outcome or solution may be unknown. Such domains are often found in law and science (e.g., (Sanders, 1994; Falkenhainer *et al.*, 1990)).

In the above example, there could be several different relationships between the locations or between the companies. The companies' locations could be in neighboring states, in the same state, in different states that both have a particular law, etc. Furthermore, the particular relationships could vary from case to case. Consider, for example, the difficulty in representing the arch shown in Figure 2.1 with some kind of feature-based representation.

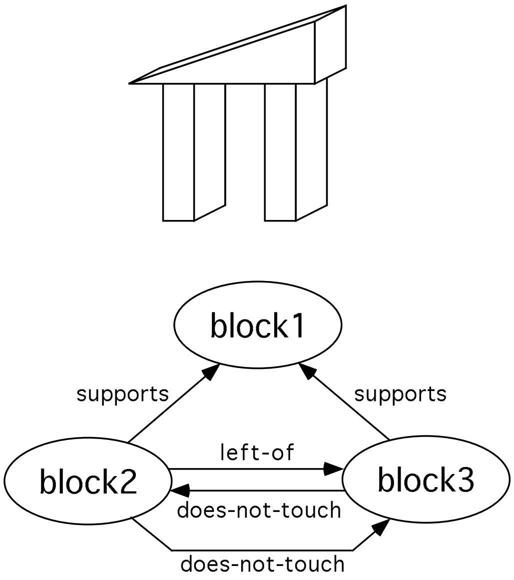

Figure 2.1: A target case (exemplar) to be classified by its structure.

In (Sanders *et al.*, 1995), we argue for *graph-structured* representations that (1) are capable of expressing the relations between any two objects in a case, (2) allow the set of relations used to vary from case to case, and (3) allow the set of possible relations to be expanded as necessary to describe new cases. Such representations can be implemented, for example, as semantic networks or lists of concrete propositions in some logic.[4] An example

---

[4] Whether the representation is implemented as a list of propositions or as a graph is immaterial, since there is a simple translation from propositional to graph representation (Nilsson, 1980).

of a graph-structured representation is the piece of a semantic network shown in Figure 2.1.

Graph-structured representations allow a system to capture a reasonably complete description of a case, i.e., all the information about a case that is potentially useful in the future. As argued in (McCartney and Sanders, 1990) case representations should be as complete as possible to maximize their usefulness. Complete representations increase a system's reasoning power: a system cannot reason with information that it does not have.

Graph-structured representations also allow the matching of relational structures to determine structural similarity between cases (see Section 2.3.1). ARCS (Thagard *et al.*, 1990) and SME (Falkenhainer *et al.*, 1990) are two analogical reasoning systems that make use of graph-structured representations for this reason. These are unlike previous feature-based models of similarity (Tversky, 1977; Ortony, 1979) in which an entity is represented as set of attribute-value pairs (*features*). As discussed above, structural relations in an analogue are difficult to capture in feature-based representations.

The Use of Graph-Structured Case Representations

Graph-structured representations have been used in several CBR systems including GREBE (Branting, 1990), a system that creates legal arguments in the domain of workers' compensation, and several CBP systems including PLEXUS, COOKIE, and CHIRON.

PLEXUS (Alterman, 1988), a CBP system for commonsense planning and plan execution (e.g., subway travel), represents background knowledge and plans in a semantic network. Nodes are linked in hierarchies by abstraction ("is-a"), partonomic ("part-of"), causal ("purpose", "goal", "reason", "outcome"), and role (filler) links. This representation supports the operations of generalization (moving up a hierarchy) and specialization (moving down a hierarchy) used to adapt plans during execution.

COOKIE (McCartney, 1993), a CBP system for cooking that integrates planning with plan execution (by people) and plan execution monitoring, uses a temporal propositional logic for case representation. This representation is graph-structured as propositions may be translated into graphs. Cases consist of a set of annotated propositions of the form (x n y t_1 t_2), where x is "occurred" or "expected", n is an identifier, y is an event or logical proposition, and t_1 and t_2 are numbers denoting a time interval. Events have the form (do s t), where s is someone and t is some action. For example, (expected ev9 (do chef1 (remove grits11 pan3)) 20.0 20.0) is a case proposition stating that expected event ev9 is the chef's removing some particular grits from some particular pan at 20 minutes into the recipe. Graph-structured case representations in COOKIE capture very detailed expected and actual actions and states, and thus allow cases to be reused in purposes unanticipated at case storage time. For example, a particular case that had the non-goal side effect of filling the kitchen with smoke could be reused in a future situation where the goal was to fill the kitchen with smoke (McCartney, 1993, p. 12).

CHIRON (Sanders, 1994), a CBP system for personal income tax planning and legal reasoning, uses graph-structured representations. A case contains a set of propositions in a temporal propositional logic (similar to that used by COOKIE) that were encoded by hand from the facts of real court cases. Because the facts can vary widely from one case to the next, a graph-structured representation can capture potentially arbitrary relations among objects in the case. For example, the representation of the court case of *Hughston v. Commissioner* contains facts such as the taxpayer Hughston was a lawyer for Shell Oil in Texas; had three children the oldest of which was ten years old; sold his old house which had a bathroom with a tile floor; bought a new house close to his new place of work, etc.

As discussed in (Sanders *et al.*, 1995), CHIRON's graph-structured representation permits the system to capture the detailed facts of its cases, and the system's algorithms for indexing, adapting, comparing, and contrasting cases exploit this rich representation. For example, every case is indexed under each of the predicates used in its case representation; thus, the system can identify as many partial matches as its rich representation language can express. Every fact can form the basis for an argument. A fact shared with a previous successful case supports an argument that the taxpayer in the current situation should succeed; a fact shared with a previous unsuccessful case suggests a possible problem. Similarly, unshared facts suggest possible distinctions between the current situation and previous cases.

CAPER uses graph-structured case-representations in which its cases are represented using a single semantic network (Chapter 3). Graph-structured representations provide the required expressivity to represent relations between objects and relations between plan actions. In the UM TRANSLOG domain (Appendix A.1), the configuration of packages with respect to vehicles in a problem's initial situation is important in retrieving a case to adapt to the target problem. Graph-structured representations allow the specification of structure queries to be processed by the PARKA Structure Matcher (Section 3.1.4). For example, if in some target planning problem an available truck and the package to be delivered are in the *same* location, it is desirable to match stored cases where this same relationship between the vehicle and package holds because such stored case will require less adaptation. The query would be "find all plans in whose initial situation $\textit{instance-of}(t, \textit{Truck}) \wedge \textit{instance-of}(p, \textit{Package}) \wedge \textit{origin}(t, x) \wedge \textit{origin}(p, x)$." CAPER also needs to represent and match on plan validations and other structural relations among subplans and plan actions.

The Cost of Graph-Structured Case Representations

The greater expressivity of graph-structured representations can come at a significant computational cost. The matching operation puts two cases in correspondence and is used for case retrieval and comparison. For feature-based representations (feature vectors and frames with atomic slot-fillers), this operation is linear in the number of features. For hybrid representations with fixed structures, it is also linear.

Using a graph-structured representation increases the computational cost of matching. If cases were represented as unlabelled graphs, matching would be the subgraph isomorphism problem, which is NP-complete (Garey and Johnson, 1979). If the graphs had node labels and no node label ever occurred more than once in a given case, matching would be linear in the sum of the number of nodes and the number of edges. In practice, the complexity of matching cases with a graph-structured representation lies somewhere between these two extremes. Some approaches taken in serial systems to reduce matching costs are multilevel partial ordering of cases (Levinson, 1991), hierarchical node descriptors (Levinson, 1991), and precomputed difference links in GREBE. As discussed in Chapter 4, CAPER makes use of the parallelism of the PARKA Structure Matcher (Section 3.1.4) to keep the cost of matching low.

With graph-structured representation, the cost of case acquisition can be high, particularly if a CBR system uses many cases. Case acquisition includes the representation and indexing of individual cases. The time cost of acquiring cases can be lower in systems using feature-based representations. In text IR (information retrieval) systems, for example, a document is represented and indexed by the words it contains and does not have to be hand-encoded nor hand-indexed; thus, case acquisition can be fully automated. Several commercial CBR "shells" provide automated indexing of case texts based on the words they contain.[5] In most CBR systems, the initial cases are derived from the experience of the system designer or some other expert, rather than from a document. As a result, a simple automated translation is not possible. The need to encode a case by hand makes case acquisition more costly.

In systems that use pure graph-structured representations, a structure needs to be created for each case during case representation. Since the facts (relations) can vary widely from case to case, representing a case is more complicated than merely filling in fixed slots in a frame. Automated methods can be used to create graph-structured representations. CAPER uses a generative planner, UM NONLIN, to seed the initial casebase (Section 3.5). The initial casebase acquisition process is completely automated because cases are not pre-indexed (Section 2.3.3).

2.3 Effective and Efficient Retrieval

Given a large number of cases encoded using a sufficiently expressive case representation scheme, an effective and efficient mechanism for case retrieval must be provided to access cases relevant to the target problem. This is a third major requirement for CBP systems. Reuse-only CBP systems, in particular, depend on retrieving a plan that is close to the target plan to produce a plan at all. Reuse-optional CBP systems can derive greater speedup if they retrieve such a plan because then less adaptation effort will be required. Retrieving a plan close to the target plan requires matching the target problem to a highly similar source case. This in turn requires flexible retrieval mechanisms that can match detailed case representations (see the previous section).

In this section, criteria for effective retrieval are discussed including various notions of *similarity* between analogues (Section 2.3.1). Techniques for making retrieval efficient are discussed including the widely used technique of case indexing for serial retrieval (Section 2.3.2). Some of the disadvantages of indexing (Section 2.3.3) and alternative approaches using parallel retrieval (Section 2.3.4), which CAPER uses, are discussed.

---

[5] E.g., Inference Corporation's CBR Express product.

Effective retrieval maximizes *recall* — the percentage of relevant cases in the casebase that are retrieved — and *precision*— the percentage of cases retrieved that are relevant. There is often a tradeoff between recall and precision (Jones, 1990). The more relevant the source plan retrieved is, the less work that will have to be done to adapt that source plan to the target problem. Efficient recall requires the fast retrieval of cases, even as the casebase grows very large.

2.3.1 Similarity

The goal of retrieval is to maximize recall and precision by retrieving source analogues that are relevant to the target analogue. Relevancy requires *similarity.* For an analogy to be more meaningful or useful, the target and source analogues (cases) should have something in common (and avoid having significant differences). Cognitive scientists have long studied what notions of similarity people use for various tasks or situations (Vosniadou and Ortony, 1989).

Several kinds of similarity are involved in human analogical reasoning including surface similarity, structural similarity, and pragmatic similarity. Two objects have *surface (semantic) similarity* if they share salient attributes such as color, shape, substance, etc. For example, oranges and tangerines have surface similarity.

Two objects have *structural similarity* if they are involved in similar systems of relations among objects. In the atom–solar system analogy, electrons and planets have structural similarity since they both orbit a central body due to an attractive force. Two stories with a common plot such as *Romeo and Juliet* and *Westside Story* also have structural similarity to each other.

Pragmatic similarity means that two entities have similar functions or achieve similar goals. The purpose of the analogy determines whether two entities are considered similar. For example, a hammer and a rock are functionally similar to each other when the goal is to pound in a nail but not when the goal is to pull out a nail. Pragmatic similarity between plans is highly desirable when doing case-based planning.

Different theories of analogical reasoning have differing claims about which criteria are (or should be) emphasized in which particular phase of analogical reasoning. For example, several psychologists present evidence that humans emphasize semantic similarity when retrieving a source analogue from a set of possible source analogues and structural similarity when comparing a retrieved source analogue to the target analogue (e.g., (Gentner and Landers, 1985; Holyoak and Koh, 1987)). Others make normative claims that structural criteria should be used by humans (and computer programs) during retrieval to obtain more relevant target analogues (e.g., (Holyoak and Thagard, 1995)).

The Structure Mapping Theory

Mapping involves putting elements of the source and target analogues (cases) in correspondence. Gentner has proposed a psychological theory of the analogical mapping process, *structure mapping*, which focuses on structural similarity (or structural isomorphism) between the target and source analogues represented as structured sets of objects, attributes, and relations (Gentner, 1983). Structurally isomorphic mappings are one-to-one and consistent. In a *one-to-one mapping*, each mapped target element maps to one and only one source element (and vice versa). In a consistent mapping, if relation $r_1 = p_1(x_1, x_2, \ldots, x_n)$ maps to relation $r_2 = p_2(y_1, y_2, \ldots, y_n)$ then argument x_i maps to argument y_i for $i = 1 \ldots n$ (where p_j is a predicate). It is possible that x_i and y_i are themselves relations.

In Gentner's model, attributes and relations are differentiated syntactically: *attributes* are unary predicates, such as *yellow(Sun)*, and *relations* are n-ary predicates (where $n > 1$), such as *revolves-around(Planets,Sun)*. The arguments of predicates can be objects (constants), such as *Sun*, or relations. If a relation has an argument that is itself a relation, it is a higher-order relation: e.g., *cause(attracts(Sun,Planets), revolves-around(Planets,Sun))*.

Gentner claims that only relations — not objects nor their attributes — should be mapped between two analogues. Her *systematicity principle* states that systems of higher-order relations are preferred over mapping lower-order relations. A potential pitfall of Gentner's approach is that it relies heavily on possibly arbitrary representational *decisions* made by a person to determine what gets mapped. For example, if the fact that the Sun is yellow is represented as an attribute *yellow(Sun)* it will *not* be mapped. But if the same fact has been represented as a relation *color(Sun,Yellow)*, it will be mapped.

In Gentner's view, semantic similarity (i.e., similarity between object attributes) and pragmatic factors (i.e., the goals of the reasoner and the purpose of the analogy) do not play a direct role in the mapping process, which is concerned with structure only. The exclusion of semantic and pragmatic factors during mapping is a subject of debate (e.g., (Holyoak, 1985; Gentner, 1989)). Gentner's theory of systematic structure mapping for analogy has been implemented using the Structure Mapping Engine SME (Falkenhainer *et al.*, 1990)).

MAC/FAC (Gentner and Falkenhainer, 1991) ("many are called: few are chosen") is a system for modeling human analogue retrieval that uses Gentner's structure mapping rules as the second stage of a two-stage retrieval process. In the first stage, the MAC component first filters out candidate source analogues that are have no surface features in common with the target. It does this by estimating *numerosity*: a content vector associated with each source is used to record the number of occurrences of a particular functor (predicate, function, and connectives) that appear in the structural description of the source. The MAC component computes a score for each source with respect to a target by computing the dot product of the target and source content vectors. Sources with scores within 10% of the highest score are passed to the FAC for further processing. Because the MAC component considers only content (functor counts) and not relations, it selects only sources that have some surface (semantic) similarity with the target, as do people (Gentner and Landers, 1985).

The FAC component of MAC/FAC, based on SME, computes a mapping between each source selected by the MAC component and the target. Because it is generally too expensive to apply SME to each source, the MAC component serves to reduce overall retrieval cost by filtering out those sources with low or no surface similarity to the target problem. The constraints of semantic similarity and structural isomorphism are thus enforced by the MAC and FAC components, respectively. The results of MAC/FAC are in accordance with the psychological data for human analogue retrieval (e.g., (Gentner and Landers, 1985)); however, like humans, MAC/FAC may miss potentially useful source analogues that are structurally similar to the target but semantically dissimilar.

The Multiconstraint Theory

Thagard and Holyoak have proposed a psychological theory of analogical reasoning that is supported by some psychological evidence and which has been tested in several computer programs (Holyoak and Thagard, 1995). They believe that the constraints or "pressures" of semantic similarity, structural isomorphism, and pragmatic centrality interact (and often compete) during analogical reasoning. Unlike Gentner, they believe that semantic and pragmatic factors play a *direct* role in analogical mapping. Like Gentner, they postulate representations of the world — objects, attributes, and relations — that are differentiated syntactically. They too favor system mappings (mappings of higher-order relations) and cite evidence from animal and child psychology that argues that the ability to do system mappings is what uniquely characterizes human analogical reasoning competence. Unlike Gentner, they believe that object attributes can be mapped during analogue mapping. Their theory is embodied in two systems: ARCS (Thagard *et al.*, 1990), a system for analogue retrieval, and ACME (Holyoak and Thagard, 1989), a system for analogue mapping. Both ARCS and ACME compare favorably with human performance on the same analogical reasoning tasks.

CAPER also represents analogues structurally and can use semantic, structural, and pragmatic similarity during retrieval and elaboration (see Chapters 3, 4, and 5 for details). For example, the physical attributes of objects in a target problem's initial situation, the relations among objects, and the target goals can all be included in a case retrieval probe. The dependency ("validation") structure of a plan can also be represented.

2.3.2 The Indexing Problem

Similarity criteria for retrieval must be made operational so that the similarity between the target case and cases in the casebase can be computed. A CBR system must search the casebase for similar cases. Much research in CBR has focused on how to index a casebase in order to make this search more efficient (Kolodner, 1993). This section surveys the technical issues of indexing. Section 2.3.3 describes some disadvantages of indexing, and Section 2.3.4 presents some alternative approaches, including CAPER's, which involve parallel retrieval.

An *index* is a data structure used to access the casebase. Here I use the term *indexing* for the creation (or updating) of an index. Recently *indexing* has been more broadly construed to mean any domain knowledge used to retrieve cases, which could include knowledge used at retrieval time (e.g., (Kolodner, 1993)). I will use the term *pre-indexing* to mean the creation (or updating) of an index prior to case retrieval time (i.e., offline or at case storage time) which serves to locate cases at retrieval time.

The purpose of indexing is to make the search for relevant cases more efficient, particularly when serial retrieval methods are used. For a real-world size casebase containing hundreds or thousands of cases and facts, it would be prohibitively expensive to check each case in the casebase for its relevancy to the target problem. Instead, an index is used to restrict the search for relevant cases to part of the casebase.

Much of the effort in building CBP systems lies in choosing an indexing scheme that can enable efficient retrieval of relevant cases from memory. This is known as the *indexing problem* in the CBR/CBP literature. An *indexing scheme* includes the data structures for representing cases, organizing the cases into a casebase, and indexing the cases. The choices for these data structures are typically interdependent. For example, a case represented as a set of attribute-value features is often indexed using a discrimination tree.

Data structures used for locating (indexing) cases include inverted files (e.g., text IR systems, (Levinson, 1991)) and discrimination trees (e.g., CHEF (Hammond, 1986), BATTLE PLANNER (Goodman, 1989)). A sample discrimination network for a casebase of cooking plans is shown in Figure 2.2. Cases are stored at the leaves of the tree. Arcs are annotated with attribute values.

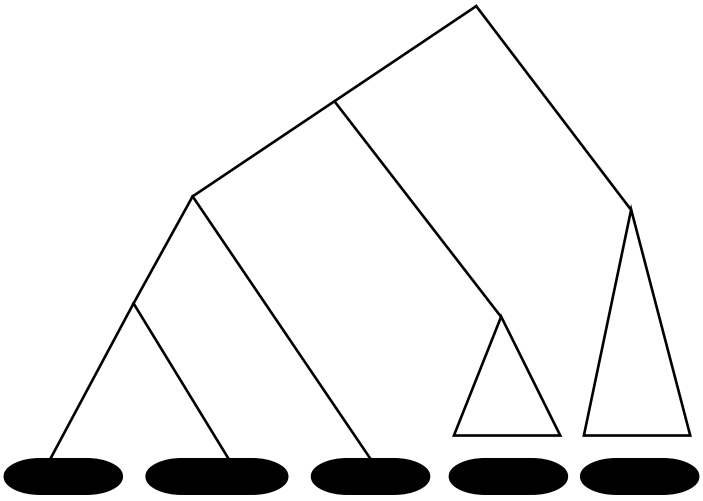

Figure 2.2: A sample discrimination tree indexing cooking plans

Indexing schemes can vary with respect to how heavily a case is indexed. Full indexing means that every feature of a case is indexed (i.e., every feature is a potential pointer for retrieval). Text information retrieval (IR) systems, for example, often index a piece of text by each word in that text (excluding some *stop* words that are poor predictors of a text's relevance). This works for small pieces of text such as abstracts of technical articles. Articles might also be partially-indexed by using (author-designated) keywords. In a CBP system, a case might be partially-indexed on several of its features and/or their abstractions.

CHEF a CBP for cooking, uses a discrimination network to index plans (cooking episodes) by their main ingredients, ingredient abstractions, type of dish (e.g., stir-fry, pasta, souffle), and taste (e.g., bland, sweet, hot). For example, a plan for making beef and broccoli stir-fry would be indexed under “beef”, “broccoli”, “meat”, and “crisp-vegetable” and would be matched given a target goal of making chicken and snow peas (Hammond, 1990). Plans are also indexed by any known failures they avoid. CHEF chooses among multiple matching plans at random. CHEF’s initial casebase consisted of 10 hand-encoded recipes.

2.3.3 The Problem with Indexing

While pre-indexing can speed up case retrieval by restricting the search for a source case to part of the casebase, it can also have several drawbacks including lower recall, lower precision, and limited utility for the casebase. The design and implementation of indexing schemes can require considerable effort. The psychological plausibility of indexing is also questionable. I will discuss these drawbacks in detail below.

The obvious tradeoff for restricting the search to part of the casebase is that relevant cases might be missed, resulting in lower recall. When partial-indexing is used, a case can only be efficiently retrieved through features by which it has been indexed or through some abstraction of them.[6] (The abstractions used need to be limited for efficiency.) Cases that share unindexed features with the target problem will not be retrieved.

It is unlikely that the designer of an indexing scheme could anticipate and provide for all of the ways in which a case might usefully be retrieved in the future. For example, one might have limited time to devote to cooking and thus desire a cooking plan for cooking any dish that takes less than 15 minutes to prepare. CHEF could not cope with such a query since it does not index plans by the time they take. It therefore would have to resort to checking potentially all cases in its casebase. One could add an index for time to each case in CHEF’s casebase, but to do this for many features would lead to a proliferation of indices and thus defeat the purpose of indexing, namely efficiency.

Precision may suffer if it turns out that too many cases have indices matching the target problem. This can happen when the indices chosen by a human designer do not discriminate well among the cases in the casebase. For example, a discrimination tree may have many cases under a single node n and few under other nodes. When features of the target problem lead the downward traversal to node n, all of the cases under n will be retrieved. This is problematic if all the cases under n are equally relevant to the target problem. In that event, all of the cases under n will need to be examined to get the most relevant one, which may be costly if there are many cases to consider.

It is difficult to design an indexing scheme that maximizes recall and precision and minimizes retrieval time. This is especially true if the casebase is to be used by multiple systems

---

[6] In such systems, it may still be possible to retrieve cases with the desired unindexed features by an expensive exhaustive search of the casebase.

for different tasks. For example, a casebase of computer equipment repair calls would be useful for both a diagnostic CBR system for repairing failures and a design CBR system for product redesign. Memory organizations used to support indexing often lack the flexibility needed to support multi-purpose memory use. Indexing structures are often domain- and task-specific. For example, a discrimination tree has a fixed ordering of indexed features by salience: the more salient features are at higher levels of the tree. In the discrimination tree of Figure 2.2, type of cuisine is the most important feature.

Creating an indexing scheme is particularly difficult when case representations are graph-structured (Section 2.2.1). Cases containing representations of complex physical structures (e.g., buildings, machined parts, molecular structures, etc.) need to be accessed for case-based design and other case-based reasoning tasks. For example, consider the task of classifying a target physical structure, shown in Figure 2.1, as an arch or non-arch using its similarity (dissimilarity) to stored training exemplars of arches (non-arches) (Winston, 1984). With a pre-indexing approach, these stored exemplars would need indices of some sort through which they can efficiently be retrieved. If these exemplars were stored under high-level indices such as "arch" or "non-arch", it would then be necessary to know whether or not the target structure is an arch or not to retrieve similar exemplars. The "archness" of the target structure, however, is what we are trying to determine. Rather than assign an abstract index to the target structure, it is better to use the representation of the structure itself to access similar exemplars in memory. Some work has been done on indexing molecular structures (e.g., (Levinson, 1991)), but the approaches often make use of the fact that substructures (e.g., component molecules) recur frequently. This does not seem to be the case, however, in other domains in which cases vary widely (e.g., legal reasoning (Sanders *et al.*, 1995)).

Some research has been done on ways to automatically build indexes. In BATTLE PLANNER (Goodman, 1989), for example, a discrimination tree is automatically induced using a variant of the ID3 algorithm (Quinlan, 1986). This produces a better-balanced tree, which enables more efficient case retrieval (i.e., $O(\log n)$ vs. $O(n)$, where n is the number of cases). The indices produced by automated techniques, however, still constrain the system at case retrieval time. Automated discrimination tree methods also need to cope with missing attributes, noise, etc. Such methods can also be misled by a nonrepresentative training set of cases.

Once the indexing scheme is designed, it must be implemented. Unless automated methods are used, the initial casebase must be indexed by hand. At case storage time, a new case must be indexed. Often this is done by hand. Manual methods would be prohibitively expensive when hundreds or thousands of cases are involved.

Finally, the psychological plausibility of indexing is questionable given evidence from psychological studies of human analogue retrieval (e.g., (Thagard and Holyoak, 1989; Waltz, 1989; Gentner and Falkenhainer, 1991)). Human analogue retrieval is fast and highly associative, with many remindings being generated. Humans are capable of finding analogies within a single domain and between domains (*cross-domain* or metaphorical analogies). Retrieval in CBR/CBP systems that use serial retrieval methods on indexed casebases does not exhibit these desirable characteristics. Psychological plausibility is a relevant criterion as CBR/CBP was originally motivated by work in human analogical reasoning and memory (Kolodner, 1993) (see Section 8.3).

2.3.4 Parallel Retrieval

To obviate the need for memory pre-indexing, several CBR systems, including CAPER, use parallel retrieval methods instead of serial ones. The use of parallel techniques means the memory does not have to be pre-indexed for efficient retrieval, even for large casebases. As memory is *not* pre-indexed, the greater flexibility at case retrieval time can be used to find a case that better matches the target problem and thus requires less adaptation.

ARCS (Thagard *et al.*, 1990) uses a parallel connectionist approach to analogue retrieval. Units representing mappings between objects, attributes, and relations are linked in a connectionist network. Excitatory links join two compatible units. Inhibitory links join two incompatible units. Two units are compatible if the mappings they represent could be part of a structurally-isomorphic mapping between a particular source analogue and target analogue. Once the links have been established, activation is given to special units whose mapped elements are semantically similar or pragmatic to the goals/purpose of the analogy. This technique gives special weight to these constraints over the constraint of structural isomorphism, an approach in accordance with some of the evidence about human analogue retrieval. A parallel constraint satisfaction algorithm is run until activation in the net settles. The source having the highest activation is chosen as the source analogue. Retrieval is competitive — a source competes with other sources — and associative, two characteristics attributed to human memory retrieval. A similar technique is used by ACME (Holyoak and Thagard, 1989) to select one of potentially several sets of mappings between a particular source and target analogue.

CAPER also uses parallel techniques to retrieve plans (or pieces of plans) from memory. Because memory is not pre-indexed, any feature of the target problem may be included as part of a retrieval description, used to form one or more retrieval probes (graphs) to be matched to CAPER's semantic network memory (Chapter 3) by the PARKA Structure Matcher (Section 3.1.4). The flexibility at retrieval time is exploited by treating retrieval description formation as a (heuristic) search process (Section 4.6).

As memory retrieval is fast, CAPER can afford to access case memory frequently and does so. It can retrieve multiple plans (or subplans) and combine them into a single plan for the target goals. Many CBP systems, in contrast, must fit a single plan to the target problem. CAPER's retrieval times have been shown to scale well to large casebases (Section 4.7).

While CAPER's retrieval algorithms can by no means be said to be identical to those used in human analogue retrieval, the *character* of retrieval in CAPER– semi-associative, parallel, frequently accessed, etc. — is more similar to that of human analogue retrieval than is that of serial CBP systems using indexed casebases. Future work, inspired by psychological experiments in long-term memory recall, includes treating retrieval itself as a planning task (Section 9.1.3).

PARADYME (Kolodner, 1988; Kolodner and Thau, 1988), a program for retrieving cases or other knowledge from a memory, also uses parallel retrieval mechanisms. Retrieval uses the massive parallelism (SIMD) of the Connection Machine (CM). Memory consists of memory organization packets (MOPs), which encode generalized and particular events and are organized in hierarchically. Retrieval is via a parallel version of a concept-refinement search algorithm based on that used in (Riesbeck, 1986), in which activation spreads bidirectionally over the hierarchy. Thus a case is found via its location in the hierarchy: no additional indexing structures are needed to match cases. The retrieval algorithm is linear in the size of the retrieval probe and therefore scales well to large casebases. PARADYME chooses among the activated cases using the sets of critical features used to annotate cases. PARADYME was used to retrieve cases for a version of the JULIA system, a case-based meal planner (Kolodner, 1989), with a casebase of 60 plans.

PARADYME is properly compared with PARKA as both implement memory with parallel retrieval mechanisms for CBR systems (JULIA and CAPER, respectively). PARKA and PARADYME are both frame-based and run on the CM, though they use different underlying data structures (see (Evett, 1994) for a comparison of these). The memory organization and concept-refinement search algorithm used in PARADYME/JULIA also differ from those of CAPER/PARKA. For example, in CAPER's memory, cases are not organized hierarchically nor are they annotated with critical feature sets. Neither CAPER nor PARADYME pre-index cases to constrain memory search. PARADYME's retrieving relevant cases, however, seems to be highly dependent on the case (MOP) hierarchy being correctly set up with respect to the domain/task at hand. Retrieval in CAPER does not use concept-refinement search with activations but instead uses the PARKA Structure Matcher to match probe graphs to the casebase, a semantic network (Chapter 4).

2.4 Effective and Efficient Adaptation

Although effective retrieval can reduce the amount of plan adaptation, typically some will be required. As a casebase is often sparse with respect to a space of possible problems in a given domain, a fourth requirement of a CBP system is having mechanisms for adapting a retrieved plan(s) to the target problem. A variety of generative techniques have been used for adaptation ranging from ad-hoc heuristics (e.g., in some reuse-only CBP systems) to provably sound and complete techniques (e.g., in some reuse-optional CBP systems). The amount of strong causal domain knowledge required also varies from system to system. The following systems perform adaptation in ways relevant to CAPER's.

PLEXUS (Alterman, 1988) is an adaptive planning system in a commonsense reasoning domain (subway travel) that uses case-based methods to adapt parts of a previous plan to a new situation on the fly. That is, using the previous plan as a guiding case, PLEXUS begins to execute the plan. Whenever a step of the old plan cannot be executed due to some difference between the old situation and the current situation, PLEXUS uses a variety of background knowledge (stored in a semantic network) to find an alternate step(s) by abstraction then specialization.

For example, it adapts its plan for riding the BART subway in San Francisco to riding the New York City (NYC) subway. In the BART plan, a ticket was purchased at a machine and used to enter *and* exit the subway platform. When re-execution of this plan is attempted in NYC, it fails when no such ticket machine is found. This is because in the NYC subway, one purchases a token that is deposited in a turnstile upon entering the subway platform (and, unlike with BART, this token is not returned by the turnstile to be collected upon exiting the subway at the destination station). Because both the entering the NYC subway and entering a theater involve buying something with which to gain admittance (a token and a ticket, respectively), PLEXUS can adapt steps from a plan for the latter to the plan for the former. PLEXUS finds the similarity between entering the NYC subway and entering a theater by generalizing the subplan "buy ticket at machine" up the is-a hierarchy to "buy ticket" and then specializing from there down to "buy theater ticket".

The techniques of abstraction and specialization are also used by CAPER to modify a failing query (Section 4.5) and to find substitutions for source plan objects in the target problem during plan mapping (Section 5.3). Such adaptation methods, however, are very knowledge-intensive and heavily dependent on the structure of the semantic network containing the background knowledge.

COOKIE (McCartney, 1993) uses a set of plan transformations to modify a retrieved plan, represented using a temporal propositional logic (i.e., graph-structured). COOKIE is somewhat conservative: permissible transformations are those where the projected outcome for the adapted case will be the same as that of the source cases. Thus transformations are based on assumptions like the fungibility of food, i.e., the same ingredients behave identically. For example, a hamburger patty can be substituted for steak in a steak-and-onions recipe because both are beef and are roughly the same shape. Transformations are applied to the source plan by permitting flexibility in the matching/unification process between propositions in the target case and those in a source case. CAPER is also able to make substitutions in plan objects during the retrieval and mapping phases. COOKIE can adapt a subplan rather than an entire source plan, if needed.

CHEF (Hammond, 1986) uses several methods to adapt and repair cooking plans. Given a source plan to adapt, CHEF first substitutes ingredients in the target plan for those in the source. CHEF uses taxonomic knowledge to determine, for example, that chicken can be substituted for beef when adapting a beef and broccoli stir-fry recipe to a chicken and snow peas stir-fry recipe, because both beef and chicken are meats. CHEF then uses domain-specific critics, each of which is associated with a particular ingredient, to modify the plan. For example, an object-specific critic associated with duck adds a duck boning step to a plan. These adaptation techniques are dependent on domain knowledge.

CHEF uses a more domain-independent approach to resolve plan failures. Abstract characterizations of plan failures are used to access TOPs (thematic organization packets), based on MOPs (Schank, 1982), which describe plan failures in the abstract. For example, a beef and broccoli stir-fry plan fails due to the broccoli being made soggy by liquid sweated from the beef during cooking. This failure is an instance of the TOP "Side Effect of a Concurrent Action Disables a Condition", where the side effect is the beef sweating, the concurrent actions are the cooking of the beef and broccoli, and the disabled condition is crisp broccoli. The applicable repair strategies associated with a TOP (e.g., Split and Reform, Alter Plan Side Effect, Adjunct Plan, etc.) are instantiated and used to modify the plan. In this example, the strategy of Split and Reform is successfully used to separate the cooking of the beef and broccoli. CHEF has 20 TOPS in its library. CHEF's TOPS and repair strategies have been compared to domain-independent techniques for plan refitting in SPA (Hanks and Weld, 1995).

PRIAR (Kambhampati and Hendler, 1992) is a case-based planning system, which, in conjunction with a hierarchical nonlinear planner such as NONLIN (Tate, 1977), supports the reuse of plans. PRIAR makes use of annotations on the hierarchical task network (HTN) of each plan produced. These annotations, collectively termed the *validation structure* of the plan, contain information about the purposes and requirements for each node in the HTN. Given a new planning problem (or a replanning problem), PRIAR retrieves a stored plan from a similar problem previously solved and computes a mapping between the old and new problems. Differences between the old and new problem can cause inconsistencies in the validation structure, which is transferred to the new problem. These inconsistencies are resolved by annotation verification and refit task specification. Both of these processes result in the modification of the target plan by PRIAR and by the planner, respectively.

Through plan reuse, PRIAR (plus NONLIN) achieved an average relative savings of 79% over planning from scratch using NONLIN alone. These savings were demonstrated in a Blocks World domain. PRIAR is sound and complete given complete causal domain knowledge and a sound and complete generative planner.

CAPER uses PRIAR-like validations to detect where a plan needs to be adapted (see Chapter 6) but lacks a full generative planner to do the necessary refitting. Unlike PRIAR, CAPER can retrieve multiple plans (or subplans) and combine them into a single plan for the target solving the target problem. This is made possible by the CAPER's flexible retrieval mechanisms and case representations. By merging parts of different plans, less planning from scratch needs to be done. Of course any harmful interactions among the merged plans must be handled. The merging and adaptation process is initially guided by the plan validation structures.

SPA (Hanks and Weld, 1995) is case-based planning system that, like PRIAR, augments a generative planner with plan-reuse capability. The plan-adaptation algorithm extends a node in a search space of partial plans by applying plan refinement operators to add/retract steps and constraints (plan dependency information) from the plan. This algorithm is sound, complete, and systematic but requires complete causal domain knowledge and a generative planner.[7] On blocks world problems, SPA's speedups (over planning without reuse) ranged from 18% to 67%. These speedups were consistently lower than PRIAR's (ranging from 34% to 99%), though this may have been an effect of the scalability of the underlying planning algorithms (Hanks and Weld, 1995).

Like PRIAR's (and CAPER's) validations, the constraints maintained by SPA during

---

[7] A planner is "systematic" if it only generates a particular state in the search space once. The underlying generative planning algorithm used by SPA (SNLP) differs from that used by PRIAR (NONLIN).

plan adaptation capture plan dependencies. SPA uses a fairly simplistic plan retrieval algorithm, matching on goals and breaking ties by counting matches in initial state facts. SPA, unlike CAPER, only adapts a single plan for a target problem.

2.5 Chapter Summary

This chapter discussed four main requirements for CBP systems and how the limitations of some previous approaches have motivated the approach taken in CAPER.

- CBR systems need large casebases.

  A larger casebase allows better coverage of the problem domain, which is crucial in reuse-only CBP systems that have limited adaptation mechanisms or weak domain theories. Real-world domains often have large problem spaces requiring thousands of cases for good coverage. Automation is needed for encoding cases. CAPER's Casebase Seeder component can automatically seed an arbitrarily large casebase.

- CBR systems need expressive case representations schemes.

  To start from a source plan close to the target plan, the CBP system must be able to compare cases at fine levels of detail. This detail must be expressible in the case representation. Pure feature-based representations (e.g., feature vectors) cannot easily express the arbitrary relations among objects in a particular case, especially when cases vary widely in their relevant facts. Graph-structured representations, such as the semantic network used by CAPER (to be discussed in the next chapter), provide greater expressivity but make case acquisition and case matching more expensive. CAPER addresses the former with its Casebase Seeder and the latter with its fast parallel retrieval mechanisms. These are described in Section 3.5 and Chapter 4, respectively.

- CBR systems need effective and efficient case retrieval mechanisms.

  To start from a source plan close to the target plan, the CBP system must be able to match cases at fine levels of detail. This requires effective similarity criteria such as surface similarity, structural similarity, and pragmatic similarity and flexible retrieval. Traditional CBR systems with serial retrieval often require pre-indexed cases for efficiency. This approach limits flexibility at case retrieval time. Because it uses fast, parallel retrieval mechanisms, CAPER does not have to pre-index cases, and a case can be retrieved through any of its features, as will be discussed in Chapter 4.

- CBP systems need effective and efficient adaptation mechanisms.

  Traditional, reuse-only CBP systems have often made do with weak domain theories and less powerful adaptation mechanisms. The ability to operate without strong domain causal knowledge can indeed be advantageous. The disadvantage of such systems is that they may fail to produce a (correct) plan if they cannot transform any source plan to the target plan. CAPER uses several adaptation methods based on plan validation structures. In addition, however, CAPER can retrieve patch or substitute plans from the casebase, as will be described in Chapter 6. This is made possible by CAPER's flexible retrieval mechanisms and case representations.

Chapter 3

CAPER's Memory

CAPER makes use of a large amount of knowledge. As discussed in Chapter 2, CBP systems require large casebases represented in such a way as to support effective and efficient case retrieval and adaptation.

Figure 3.1 shows the basic kinds of data used by CAPER: its Memory (Knowledge Base), Working Data, and Domain Knowledge. Working Data is a collection of data structures that contain temporary data used by CAPER. Often this data contains meta-knowledge about information in CAPER's Memory. These data structures will be described in subsequent chapters along with the components of CAPER that use them.

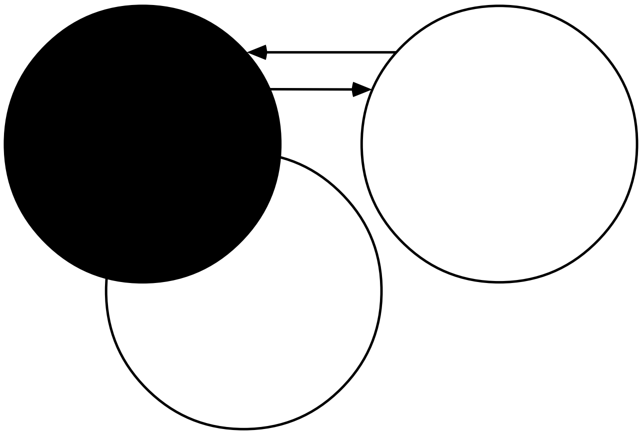

Figure 3.1: Kinds of data used by CAPER.

Domain knowledge consists of persistent domain-specific data in several forms: procedures and methods; parameters and heuristics; and concepts and individuals. Domain-specific procedures, methods, parameters, and heuristics will also be described along with the components of CAPER that use them. Domain concepts and individuals are part of CAPER's Memory, the focus of this chapter.

The *Memory*, or "Knowledge Base" (KB), is CAPER's store of *persistent* knowledge including concepts, individuals, and cases.[1] Some of the knowledge in the Memory is domain knowledge. CAPER's *Casebase* (CB) is that part of the KB used to describe particular cases, plans, and actions. All knowledge in CAPER's Memory is represented as a single, highly interconnected semantic network.

Section 3.1 describes PARKA, the frame-based, knowledge representation system used to implement CAPER's Memory. PARKA frames are used to implement the semantic network. Using PARKA language, I designed specific representation schemes for concepts, individuals, and episodes (cases, plans, and actions), described in Section 3.2, Section 3.3, and Section 3.4, respectively. These representation schemes are fairly general and thus usable for multiple domains. Examples will be given from the various domains being used to test CAPER (Appendix A).

CAPER gets its knowledge in two ways. Conceptual knowledge is hand-encoded by a person: a knowledge/domain analyst/engineer. Episodic knowledge is initially generated by the Casebase Seeder using the UM NONLIN planner. This process is described in Section 3.5. As CAPER solves new problems, episodic knowledge is added to the KB (Section 6.5).

3.1 Memory Implementation Using PARKA

CAPER's Memory includes persistent conceptual and episodic knowledge stored in a single, highly interconnected semantic network, which is implemented using the high-performance, parallel, frame-based knowledge representation system PARKA (Evett *et al.*, 1994; Andersen *et al.*, 1994; Evett, 1994; Kettler *et al.*, 1995).[2] PARKA uses parallelism to perform extremely rapid property inheritance and other kinds of inferences. PARKA's current design focus is on speed at query time, rather than on speed at update time (when a frame or assertion is added or deleted). PARKA was originally implemented for the massively-parallel Connection Machine (CM) (Hillis, 1985) and is being extended to a wide variety of other parallel supercomputers.

A PARKA knowledge base (KB) is frame-based. Frames can be used to implement a semantic network. In PARKA, a KB can in fact be viewed equivalently as a semantic network or a frame system. A frame x with its p-slot filled by a frame y, can be viewed as a node in a semantic network x with a p-link to node y. The frame "view" of a PARKA KB emphasizes the encapsulation of data, i.e., the facts about an entity are collected in

---

[1] The capitalized "Memory" refers to CAPER's memory, its knowledge base (KB). Recall that "memory" as used in this document refers either to a KB or human memory, not physical computer memory.

[2] PARKA is a research project in the Parallel Understanding Systems Group at the University of Maryland at College Park, under the direction of Prof. James Hendler. The original version of PARKA was developed by Matthew Evett and Lee Spector. The current version of PARKA (Version 3) was developed by William Andersen, Brian Kettler, and Sean Luke. PARKA is currently being extended to additional parallel platforms by William Andersen, Matthew Evett, and Kilian Stoffel.

a frame representing that entity. The semantic network view of the KB emphasizes the interconnectedness of knowledge, i.e., a node representing an entity is connected to nodes representing other related entities. The benefits of frames and semantic networks in terms of expressivity, economy, etc. are described elsewhere (e.g., (Lehmann, 1992)).

PARKA provides a language for representing entities (concepts and individuals) and facts about them (the PARKA language). Inference and query mechanisms are provided for accessing implicit and explicit information in the KB. The PARKA system provides mechanisms for user interaction and KB maintenance. These components of PARKA are described briefly in the following sections. The emphasis here will be on PARKA's primary features, particularly ones taken advantage of by CAPER.

3.1.1 The PARKA Language

This section briefly describes the main representational features available in PARKA. Details can be found in (Kettler *et al.*, 1995). The KB in Figure 3.2 will be used as a running example. It is based on a sample car construction domain (Appendix A.2) used for an earlier version of CAPER. The figure shows a portion of the semantic net representing facts about a particular car, *Car-1*. Entities from the "world" are represented by categories and instances in PARKA.

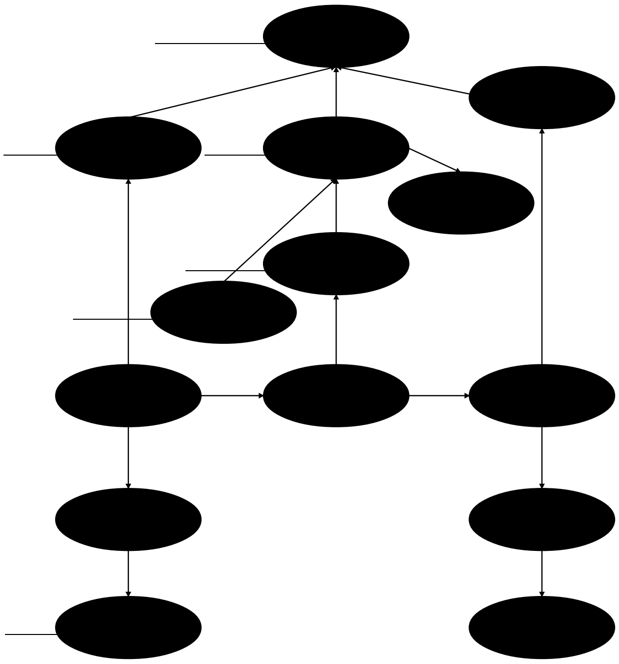

Figure 3.2: A part of the net representing a particular car, Car-1. (Nodes with underlined labels are categories. Nodes with plain labels are instances. Links represent assertions.)

Categories

Categories represent sets. Categories thus correspond to concepts defined extensionally. For example, the frame *Car* represents the set of all cars. Categories can have subcategories and supercategories, e.g., *Car-Engine* has supercategory *Physical-Object* and subcategory *4-Cyl-Engine.*

Categories are related to their supercategories by *isa* links: if there is an *isa* link from c_1 to c_2, then c_2 is a supercategory of c_1. The *isa* link corresponds to the subset relation between sets: if c_1 is a subcategory of c_2, then set represented by c_1 is a subset of the set represented by c_2. Thus, the set of 4 cylinder engines is a subset of the set of engines.

There is a frame in the KB for each category that is defined. Category (frame) names typically begin with a capital letter. Categories are indicated in this document's figures by ovals with underlined text labels.

Instances

Instances represent particular individuals. For example, *Car-1* represents a particular car, designated "Car-1". The color red is represented by the instance *Red.* An instance is a member of a category.

Instances are related to their categories by *instanceOf* links: if there is an *instanceOf* link from instance i to category c, then i is an instance (member) of c. In other words, the entity represented by i is an element of the set represented by c. For example, *Car-1* is an instance of category *Car*, i.e., *Car-1* is a particular car.

There is a frame in the KB for each instance that is defined. Instance (frame) names typically begin with a capital letter and, in CAPER, often end in a numerical suffix (e.g., *Car-1*). Instances are indicated in this document's figures by ovals with plain (i.e., non-underlined) text labels.

Predicates

Predicates are used to define relations among entities. For example, the predicate *colorOf* might define the "color of" relation that relates things to colors, e.g., *Car-1* to *Red.* Currently only binary relations are supported in PARKA. A predicate has an associated domain and range that specifies what is being related to what. For example, the predicate *colorOf* has domain *Physical-Object* and range *Color* indicating that this predicate relates *instances* of category *Physical-Object* to *instances* of category *Color* (i.e., a physical object can have a color).

There is a frame in the KB for each predicate that is defined. Predicates are thus first-class objects because one can express properties about them (e.g., their domain, range, associated inference method, etc.). Predicate names typically begin with a lowercase letter. Predicates are indicated in this document's figures by labels on links (arrows). The link goes from an instance of the predicate's domain (category) to an instance of the predicate's range (category).

Assertions

Assertions are used to specify instances of relations, which are represented by predicates. A binary assertion is of the form $p(x, y)$, where x and y are arguments (entities represented by categories or instances) and p is a binary predicate. An assertion corresponds to a p-link from x to y or, alternatively, frame x's p-slot being filled by a value of frame y. For example, the fact that *Car-1* is red is represented by the assertion *colorOf(Car-1,Red)*.

There is *no* frame in a KB that corresponds to an assertion: i.e., assertions are *not* reified in PARKA. Assertions are indicated in this document's figures by labeled links (arrows).

Other Language Features

Other PARKA language features especially relevant to CAPER include the following:

- **Multiple slot fillers**

  A slot can be filled with multiple values.

- **Immediate value slot fillers**

  A slot can be filled with values other than frames including integers, floats, strings, and Lisp symbols.

- **Set and vector slot fillers**

  A slot can be filled with a single object that is a set (Lisp list) or vector (Lisp vector) of frames or immediate values.

- **Slot inverses**

  A predicate p can have a defined inverse predicate p_i. When $p(a, b)$ is asserted (retracted), PARKA will automatically assert (retract) $p_i(b, a)$.

- **Update methods**

  A Lisp function may be associated with a predicate and will be executed whenever an assertion is made for that predicate.

3.1.2 Inference Methods

PARKA's inference methods allow slot values to be calculated, rather than explicitly specified. If no explicit value is specified for a frame's slot, PARKA may be able to infer a value from those values specified for other frames. The inference methods in PARKA are inferential distance ordering (IDO) inheritance, transitive inference, and transfers-through inference. In addition, inference methods can be composed, and predicates can be specified as noninheritable, meaning that slots for them will not be inherited using the default method (IDO). The first two of these methods are used most often by CAPER and are described below. Each predicate has an associated inference method, the default of which is inferential distance ordering.

Inferential Distance Ordering Inheritance

Inferential Distance Ordering (IDO) inheritance allows slot values (for predicates specified as using IDO inheritance) to be inherited from other nodes (frames) along *isa* and *instanceOf* links. An (explicit or inferred) value can be inherited by an instance over a single *instanceOf* and zero or more *isa* links. Thus, an instance can potentially inherit a value from its immediate category or supercategories of its immediate category. In Figure 3.2, for example, the instance *4-Cyl-Engine-1*, which lacks an explicit value for its potential *colorOf* slot, can inherit a value (*Silver* from its category *Car-Engine* over one *instanceOf* link and one *isa* link.

Conflicts can arise when a frame inherits from multiple frames. To handle this problem of multiple inheritance, PARKA uses a variant of Touretsky's IDO algorithm (see (Evett, 1994) for details) that uses the relative topological depths of the frames that can be inherited from. Topological depths are calculated in a parallel downward traversal of the *isa*/*instanceOf* hierarchy.

Transitive Inference

Transitive inference allows slot values to be inferred from other nodes (frames) along transitive links. A transitive link is one whose predicate has been specified as transitive. If predicate p_t is asserted to be transitive over predicate p, then the relation for p_t specifies the transitive closure of the relation for p. In Figure 3.2, for example, if a predicate *tpartOf* was defined as transitive over *partOf*, then PARKA could infer the assertion *tpartOf(Carburetor-1,Car-1)* from the assertions *partOf(Carburetor-1,4-Cyl-Engine-1)* and *partOf(4-Cyl-Engine-1,Car-1)*.

3.1.3 Queries

PARKA has three kinds of queries: simple queries, recognition queries, and structure queries. The first two are described in this section. Structure queries are described in the following section.

Simple Queries

Simple queries are of the form "find all assertions (explicit or inferred) that match *p(x,y)*", where x is a frame or variable; y is a frame, variable, or immediate value; and p is a predicate (frame). In Figure 3.2, for example, the following are simple queries:

1. Is Car-1 red?: *colorOf(Car-1,Red)* (i.e., x and y are constants)
2. What color is Car-1?: *colorOf(Car-1,?C)* (i.e., x is a constant, y is a variable)
3. What things are red?: *colorOf(?C,Red)* (i.e., x is a variable, y is a constant)
4. What things are which colors?: *colorOf(?T,?C)* (i.e., x and y are variables)

Queries such (1) and (2) are bottom-up queries: they are processed by upward traversal of links from a frame. Query 2, for example, is processed by examining the *colorOf* slot for *Car-1*. If a value is found, it is returned. Otherwise, the category(ies) of *Car-1*, *Car*, is visited and its *colorOf* slot checked for a value to be returned. In serial or parallel, bottom-up queries can be processed in time $O(d)$, where d is the depth of the *isa*/*instanceOf* hierarchy (d is typically $O(\log n)$), where n is the number of nodes in the net).

Queries such as (3) and (4) are top-down queries: they are processed by downward traversal of links from the root. Query 3, for example, is processed by returning the (explicit or inherited) value of the root node's *colorOf* slot and by recursively checking the subtree's of this node. Top-down queries can be processed in parallel with p processors $O(\frac{nd}{p})$. Given a sufficiently large number of processors (i.e., $p > n$), top-down queries can be done in $O(d)$ in parallel.[3] Serial processing of top-down queries is $O(nd)$ (typically $O(n \log n)$), for a naive implementation.

Recognition Queries

Recognition queries are of the form "find all frames x such that $c_1 \wedge c_2 \wedge \ldots \wedge c_m$". Here, c_i is a unary constraint (a constraint involving exactly one variable) of the form $p_i(x, v_i)$ or $p_i(v_i, x)$, where p_i is a predicate frame, x is a variable, and v_i is a constant (i.e., a frame or immediate value). The variable x must be the *same* in every constraint c_i. In Figure 3.2, for example, a recognition query to find all red cars would be "*colorOf(?T,Red)* $\wedge$ *instanceOf(?T,Car)*".

In the worst case, a recognition query decomposes into m top-down queries, one for each c_i. PARKA can compute these queries in time

$$O(\sum_{k=1}^{m} \frac{nd}{p})$$

or $O(md)$ if $n > p$.[4] A naive serial algorithm would take, on average, time $O(mdn)$ for such queries.

3.1.4 PARKA Structure Matcher

Structure queries are similar to recognition queries. Unlike recognition queries, however, structure queries can contain several *different* variables. Structure queries are of the form "find all subgraphs in the semantic net (graph) that match a probe graph".

The syntax for structure queries is "$c_1 \wedge c_2 \wedge \ldots \wedge c_m$", where c_i is a constraint. Each c_i has the form $p_i(x_i, y_i)$, where p_i is a predicate frame; x_i is a variable or constant (a frame or immediate value); and y_i is a variable or constant. If both x_i and y_i are variables, c_i is a

---

[3] This analysis is somewhat over-simplified as there are additional costs due to communications overhead, etc. See (Evett, 1994) for details.

[4] This analysis is somewhat over-simplified as there are additional costs due to communications overhead, etc. See (Evett, 1994) for details.

binary constraint. Otherwise, it is a unary constraint. Variables correspond to nodes in the probe graph (variables are not allowed for edges). Constraints correspond to edges in the probe graph. A unary constraint is an edge connecting a node corresponding to a constant to a node corresponding to a variable. A binary constraint is an edge connecting two nodes that correspond to two variables.

In Figure 3.2, a structure query to "find all 4 cylinder engines that are part of red cars" corresponds to the probe graph in Figure 3.1.4. This query can be written as "*instanceOf(?Engine,4-Cyl-Engine* $\wedge$ *instanceOf(?Car,Car)* $\wedge$ *colorOf(?Car,Red)* $\wedge$ *partOf(?Engine,?Car)*". The first three clauses are unary constraints. The fourth clause is a binary constraint.

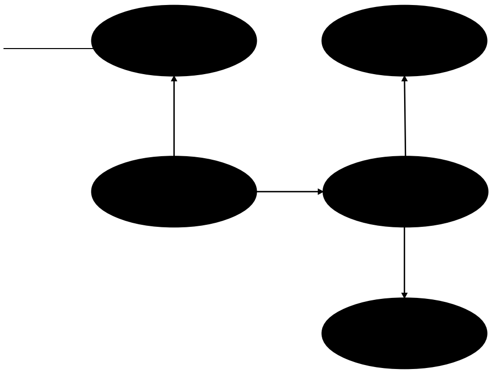

Figure 3.3: A probe graph for a sample structure query: find red cars with four cylinder engines. (Query variables are prefaced with a "?".)

When processing a probe graph, the PARKA Structure Matcher looks for an exact match. If it finds one, it returns all sets of variable bindings, one set for each subgraph matched. If no exact match is found, NIL is returned. The extension of the algorithm to support partial matching and inexact matching is being investigated. A partial match is a probe that matches part of the subgraph. An inexact match is a modified probe that matches a subgraph. Currently the relaxation of the probe graph is under control of CAPER, not PARKA (Section 4.6).

Some empirical results for retrieval using the PARKA Structure Matcher on CAPER knowledge bases are presented in Section 4.7. For an analysis of the computational complexity of the PARKA Structure Matcher, see (Andersen *et al.*, 1994).

3.1.5 The PARKA System

PARKA's base ontology (a set of concepts) initially contains top-level concepts used by PARKA including meta-categories (e.g., *Category*, *Predicate*, *Value*, etc.

The PARKA system has a serial front-end and a parallel back-end. The serial front-end is currently written in Common Lisp and runs on an Apple Macintosh (Quadra 800).

A SIMD implementation in *Lisp of the current version of PARKA (Version 3) currently exists for the Thinking Machines Corporation CM-5. Previous versions of PARKA ran on the CM-2. Currently a MIMD back-end implementation in C is being done for the IBM SP-2 and other parallel supercomputers.

The front-end can be run in stand-alone mode to enable testing of application programs such as CAPER without requiring the back-end parallel machine to be available. In normal mode, the parallel back-end is available and queries that make sense to do in parallel are routed from the front-end client to the back-end server, where they will be processed. The front-end stores the entire KB and updates are made there.

3.2 Concepts

Conceptual knowledge is part of CAPER's Memory (KB), which contains persistent knowledge represented as a semantic network and implemented as frames in PARKA. CAPER uses conceptual knowledge to reason at more abstract levels and to organize the episodic knowledge (plans and cases), described in Section 3.4. Concepts, the subject of this section, are represented by PARKA categories. Concepts have instances, represented by PARKA instances. These instances are *individuals* and are described in Section 3.3.

Facts relate concepts to other concepts and individuals. Facts are represented by PARKA assertions. In CAPER, facts and concepts are of three basic kinds: general, planning, and domain. They include general facts such as "a physical object can have a location", planning facts such as "a plan has an ordered list of actions", and domain facts such as "tankers can carry liquids." Facts can be explicit or inferred via one of PARKA's inference mechanisms. General and planning concepts are described in the following sections. Domain concepts include objects, actions, and constraints for a particular domain, described in Appendix A.

A set of concepts and their associated facts is called an *ontology*. All concepts, individuals, and facts are stored in the same, highly interconnected KB. CAPER's base ontology, built on top of the predefined (built-in) PARKA ontology, contains domain-independent concepts (general and planning concepts). Each domain has a domain ontology which builds on the base ontology. The sizes for these hand-built ontologies are shown in Table 3.1.

3.2.1 General Concepts

General concepts include physical objects (*Person*, *Place*, *Object*), physical properties (*weight*, *location*, *color*), abstract concepts (*Sequence*, *memberOf*), time-related concepts (*Event*, *Episode*, *Situation*), and meta-data concepts (*Attribute-Value*, *Situated-Slot*). General concepts include predicates (*isa*, *color*) and entities (*Sky*).

3.2.2 Planning Concepts

Planning concepts include *Action*, *Condition*, *Effect*, *Plan*, and *Goal*. Planning knowledge can be abstract or episodic. Abstract planning knowledge is of two kinds: domain-

Ontology	**Frames**	**Cats.**	**Instances**	**Predicates**	**Assertions**
PARKA	117	38	79	45	719
CAPER	399	109	290	118	2415
UMT	517	233	284	140	3784

Table 3.1: Incremental sizes of PARKA predefined, CAPER domain-independent, and UM TRANSLOG domain ontologies. (Frames include categories, individuals, and predicates. "Values" are non-frame, immediate values (strings, symbols, etc.). Each ontology is built upon the ones above it. Ontologies do *not* include cases.)

independent and domain-specific. Domain-independent planning knowledge includes concepts such as *Action* and *Effect* and facts such as "actions can have effects." Domain-specific planning knowledge includes knowledge about stereotypical domain actions, i.e., their default conditions, effects, roles, etc.

Figure 3.4 shows some of the planning concepts that are part of CAPER's base ontology. This figure shows an action category, *P-Load-Pkg*, which represents a set of actions that involve loading packages into vehicles. This action is considered *primitive* because it can be executed directly by some agent without the need for further decomposition by the planner. By convention, primitive actions are represented by categories whose names begin with a "P-". The nodes shown in Figure 3.4 that are instances (i.e., the nodes at the bottom of the figure with text labels that are *not* underlined) represent particular episodes, part of CAPER's episodic planning knowledge (see Section 3.4).

3.3 Individuals

In CAPER, individuals — particular entities —are represented by PARKA instances. These entities can be tangible (e.g., a particular truck) or intangible (e.g., a particular plan). They can be real (e.g., an executed, instantiated action) or imagined (e.g., a unexecuted, instantiated action).

For example, consider the portion of CAPER's Memory shown as a semantic network in Figure 3.5. (This figure and the remaining figures in this chapter are taken from the UM TRANSLOG domain (Appendix A.1).) Figure 3.5 shows nodes representing three individual things in the world *Pkg-1.1* (a regular package), *Truck-1* (a regular truck), and *City1-CL1* (customer location #1 in city *City-1*). Their categories and supercategories via the *instanceOf* and *isa* links, respectively, are also shown. The following facts are also represented:

- Regular-type packages are compatible with (i.e., can be carried by) regular-type trucks.
- *Pkg-1.1* is located at *City1-CL1* in situation *Situation-1*.

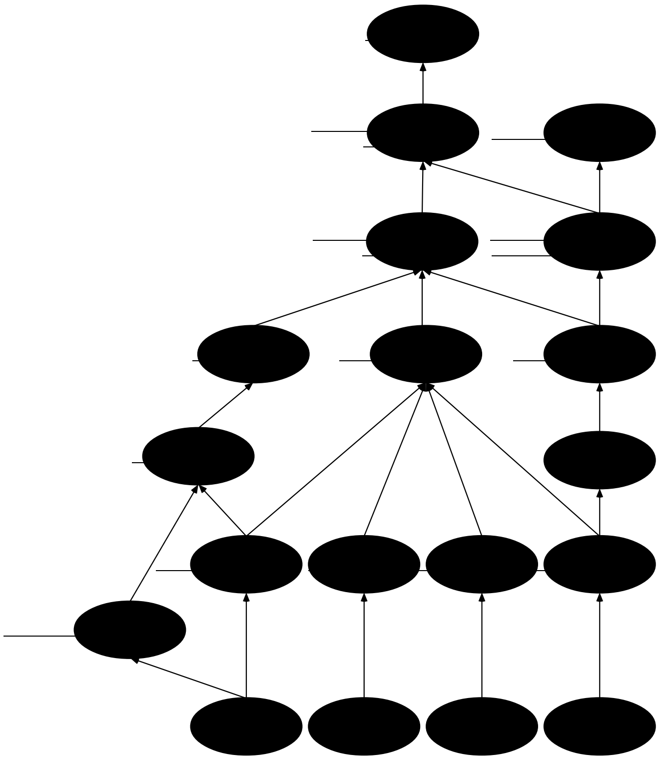

Figure 3.4: A part of the net representing planning concepts. (Nodes with underlined labels are categories. Node with plain labels are instances. Links represent assertions. Only the *isa/instanceOf* hierarchy is shown here. In this and subsequent figures, some nodes and links have been omitted for clarity.)

- *Truck-1* is located at *City1-CL1* in situation *Situation-1*.

The last two facts are represented using *situated slots*. A situated slot is a slot filled by one or more *situated value frames* such as *Situated-Value-1* and *Situated-Value-2*. In the absence of a context mechanism in PARKA, situated value frames are a representational convention used to express that a slot has potentially different values, depending on the context. Each situated value frame has a *situation* slot, filled by an instance of *Situation*, and a *value slot*, filled by an appropriate value, e.g., *Situated-Value-2* has its situation slot filled by *Situation-1* and its value slot filled by *City1-CL1*. *Truck-1* has multiple situated value frames in its *location* slot as its location changes in different situations (i.e., its initial location may be different in each case's initial situation).

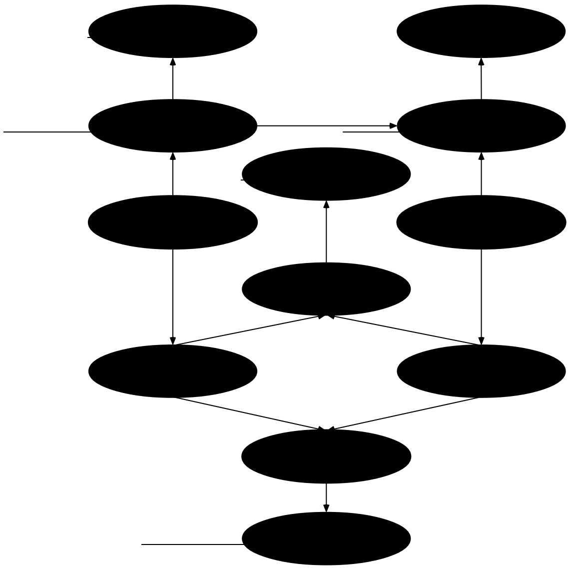

Figure 3.5: A part of the net representing three individuals: Pkg-1.1, Truck-1, and City1-CL1.

3.4 Episodes

Episodic knowledge includes knowledge about particular temporal occurrences (events, actions, plans, etc.) and the particular objects associated with them. Episodic knowledge involves individuals (Section 3.3) of various types including objects (*Truck-1*, *Pkg-1.1*), primitive actions (*P-Open-Door-1*), plans (*Plan-Located-At-1.3*), cases (*Case-Make-Deliveries-1*), and situations (*Situation-1*).

In Figure 3.4, for example, a particular, individual occurrence of loading a package is represented by the instance (frame) *P-Load-Pkg-1.1*, an instance of the action category *P-Load-Pkg*. I use a numerical suffix for the names of frames representing individuals (objects, episodes, etc.). The name root typically indicates the category. A numerical suffix of the form *c.i*, indicates the *i*th object/episode/etc. of case *c*.

While the representations of episodic knowledge were engineered specifically for CAPER, they turn out to resemble those used to represent events in a particular theory of autobiographical memory motivated by cognitive psychology experiments (Barsalou, 1988). CAPER's episodes, however, are not stored chronologically as are events in Barsalou's model. Some of the CAPER's representations were inspired by those in the CYC KB (Lenat and Guha, 1990).

Figure 3.6 lists the kinds of episodic knowledge stored in CAPER and shows some of the relationships among them. The following sections describe the representations for episodic knowledge in detail.

Figure 3.6: Kinds of episodic knowledge in CAPER

3.4.1 Cases

Cases are a kind of episodic knowledge. In CAPER, cases are subgraphs of the semantic network. This kind of graph-structured representation (see Section 2.2.1) enables the encoding of detailed facts that can vary from case to case and allows structural similarity between cases to be determined at case retrieval time. Cases are not stored in a case hierarchy, i.e., a case does not have an abstraction link to another cases (such as in discrimination net) and are not pre-indexed (see Section 2.3.2).

In CAPER a case consists of the facts holding in the initial situation, goal(s) to be achieved, the plan(s) generated to achieve the goals, meta-planning information, and plan execution information. Meta-planning information includes the plan creator (e.g., a human, CAPER, or some other planning system), planner version (if the plan was created by a planning system), etc.

Figure 3.7 shows a semantic network representation of a portion of a particular case, *Case-Make-Deliveries-1*, that specifies the following:

- *Case-Make-Deliveries-1* is an individual case (i.e., an instance of *Case*), with the single goal *Make-Deliv-1-Goal-1*.
- *Make-Deliv-1-Goal-1* is an individual goal (i.e., an instance of *Goal*) which represents the single goal of this case (i.e., delivering a package) (See the details in Figure 3.10).
- *Make-Deliv-1-InitSit* is an individual situation (i.e., an instance of *Situation*) and represents the initial situation of this case.
- *Plan-Located-At-1.3* is an individual plan (*E-Plan*) that was created by UM NONLIN planner.

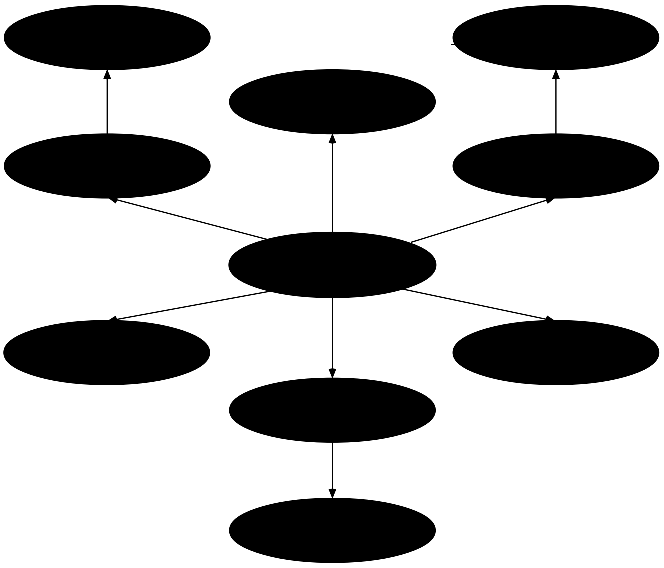

Figure 3.7: A part of the net representing the Case Case-Make-Deliveries-1.

3.4.2 E-Plans

E-Plans (episodic plans) are a kind of episodic knowledge. They are plans that have been wholly or partially instantiated. An e-plan consists of the facts holding in the initial situation (i.e., before any actions are executed), the goal(s), the hierarchical task network (HTN) generated, plan validations, meta-planning information, and execution information. The HTN is described in the following section. Plan goals and validations are described in Section 3.4.4 and Section 3.4.6, respectively.

In *Case-Make-Deliveries-1*, regular-type truck *Truck-1* is initially at customer location *City1-CL1* and regular-type package *Pkg-1.1* is initially at *City1-CL1*. The goal is to deliver *Pkg-1.1* to customer location *City1-CL2*. The following are the e-actions of the plan *Plan-Located-At-1.3*, generated by UM NONLIN(see Section 3.5), that accomplished this goal (see Appendix C.2 for details about this plan):

1. **Do-Nothing-1.1** (Do-Nothing) — do nothing[5]
2. **Do-Nothing-1.2** (Do-Nothing) — do nothing
3. **P-Collect-Fees-1.1** (P-Collect-Fees) — collect fees for package *Pkg-1.1*

---

[5] An action of type *Do-Nothing* is a null action: nothing is to be done by the agent executing the plan. The "do nothing" actions are produced in plans generated by UM NONLIN when it expands certain nodes, as specified for the domain. These null actions are left in the plan when it is encoded for CAPER because they function as place holders and have various plan validation information associated with them.

4. **Do-Nothing-1.3** (Do-Nothing) — do nothing
5. **P-Open-Door-1.1** (P-Open-Door) — open cargo door of truck *Truck-1*
6. **P-Load-Package-1.1** (P-Load-Package) — load package *Pkg-1.1* into *regular* truck *Truck-1*
7. **P-Close-Door-1.1** (P-Close-Door) — close cargo door of truck *Truck-1*
8. **P-Move-Vehicle-1.1** (P-Move-Vehicle) — drive truck *Truck-1* from *City1-CL1* to *City1-CL2* using local roads in city *City1*.
9. **P-Open-Door-1.2** (P-Open-Door) — open cargo door of truck *Truck-1*
10. **P-Unload-Package-1.1** (P-Unload-Package) — unload package *Pkg-1.1* into *regular* truck *Truck-1*
11. **P-Close-Door-1.2** (P-Close-Door) — open cargo door of truck *Truck-1*
12. **Do-Nothing-1.4** (Do-Nothing) — do nothing
13. **P-Deliver-1.1** (P-Deliver) — hand package *Pkg-1.1* to customer

Following the e-action's name is its action type (Section 3.4.5).

3.4.3 Hierarchical Task Network (HTN)

The hierarchical task network (HTN) is a tree that captures the structure of a plan in terms of its component plans (subplans) and actions. The interior nodes of this tree are nodes are subplans (e-plans), and the leaf nodes are primitive actions (e-actions).[6] HTN nodes (and their corresponding semantic network nodes) are linked by *subplan* and *action* links. The former link a plan to its subplan(s). The latter link a plan to its actions.

Figure 3.8 shows part of the HTN for plan *Plan-Located-At-1.3* Details can be found in Appendix C.2.2. Part of the semantic net representation for this HTN is shown in Figure 3.9.

Several terms are used to describe relationships among **HTN nodes**. The **root** of an HTN is the top-level e-plan node, e.g., *Plan-Located-At-1.3* is the root node in the above HTN. A **child** of an HTN node n is a node n_c that is connected to n by a *subplan* or *action* link. The **parent** relation is the opposite of the child relation. Thus if n_c is a child of n, n is the parent of n_c. The **descendant** and **ancestor** relations are the (proper) transitive closure of the child and parent relations, respectively. In above HTN, for example, e-plan *Plan-Pkg-Loaded-1.1* is a child of e-plan *Plan-Pkg-Loaded-1.2*. The latter is the parent of the former. The descendants of *Plan-Pkg-Loaded-1.2* include *Plan-Pkg-Loaded-1.1* and all of the nodes below it in the HTN. The ancestors of *Plan-Pkg-Loaded-1.2* include plan *Plan-Loaded-Val-1.1* and all of the nodes above it in the HTN.

---

[6] HTN nodes should not be confused with the PARKA semantic network nodes used to represent them.

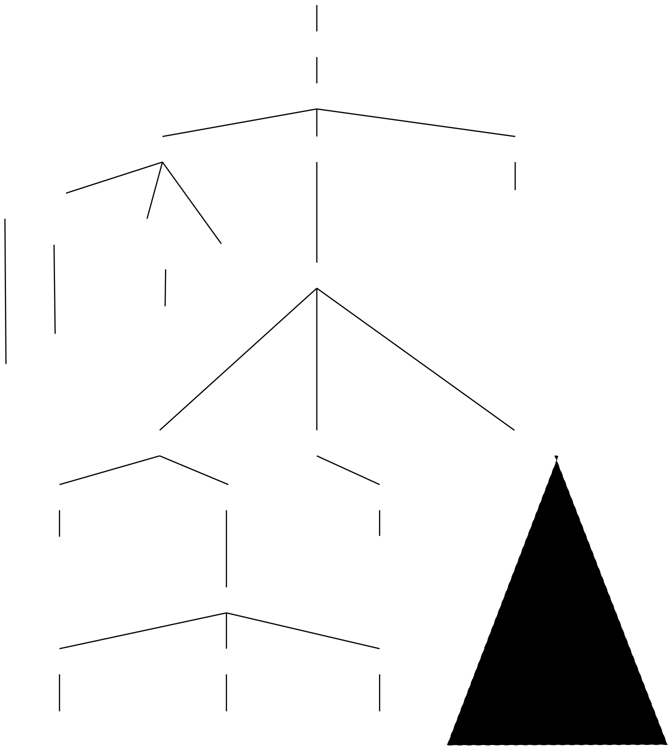

Figure 3.8: The HTN of Plan-Located-At-1.3. (Each label corresponds to an HTN node, an e-action (bold label) or an e-plan (plain label). E-plans are comprised of e-eactions. A link connects a parent HTN node with its child(ren). The subplans below *Plan-Not-Pkg-Loaded-Top-1.1*, omitted in the figure due to space constraints, are analogous to those below *Plan-Pkg-Loaded-Top-1.1*.)

Figure 3.9 shows a semantic network representation of a portion of HTN for *Plan-Located-At-1.3* (Figure 3.8) that specifies the following:

- *Plan-Located-At-1.3*is an individual e-plan used to successfully achieve the goal *Make-Deliv-1-Goal-1* (Figure 3.10), which is the goal of case *Case-Make-Deliveries-1* (shown in detail in Figure 3.7).
- *Make-Deliv-1-Goal-1* is an individual goal (i.e., an instance of *Goal*) which represents the single goal of this case (i.e., delivering a package) (shown in detail in Figure 3.10).
- *Plan-Located-At-1.3* has e-plan *Plan-Transported-1.1*, an e-plan, as one of its subplans. *Plan-Located-At-1.3* is the root of the HTN.
- *Plan-Pkg-Loaded-1.1* has (primitive) action *P-Load-Pkg-1.1*, an e-action (shown in detail in Figure 3.11).

- Etc...

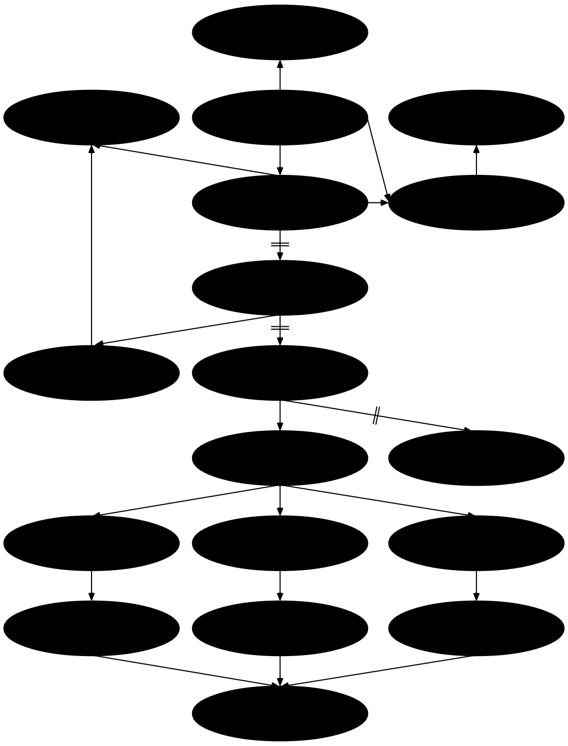

Figure 3.9: A part of the net representing the HTN rooted at plan Plan-Located-At-1.3. (A semantic net node in this diagram corresponds to an HTN node in Figure 3.8. Hash marks on links indicate that intervening nodes are not shown.)

PARKA's slot inverse is used extensively here. For example, the predicate *subplan* has the predicate *subplanOf* as its inverse. Thus when *subplan(Plan-Pkg-Loaded-1.2,Plan-Pkg-Loaded-1.1)* is asserted by CAPER, *subplanOf(Plan-Pkg-Loaded-1.1,Plan-Pkg-Loaded-1.2)* gets asserted automatically by PARKA.[7]

The predicate *subplan* is also transitive (Section 3.1.2). Thus if x is a subplan of y and y is a subplan of z, it will be inferred via PARKA's transitive inheritance mechanism that x is a subplan of z.

---

[7] A recent extension to PARKA that could be exploited by CAPER would be inheritance over the *subplan* links: i.e., slots could be inherited by a plan from its parent plan(s).

3.4.4 Conditions, Goals, and Effects

Conditions, goals, and effects are propositions which describe world states. A condition describes a state (or set of states) that should exist before an action is executed. A goal describes a state that is to be achieved by a plan. An effect describes a state that exists after an action's execution.

For example, the proposition *color(Truck-1,Red)* describes the world states in which Truck-1 is red in some possible world. A corresponding goal is "Make the truck red." A corresponding condition is "The truck is red." A corresponding effect is "The truck is made red."

Conditions, goals, and effects have a ce-type (condition/effect type), modality, and roles. A ce-type is a category and corresponds to the argument of a proposition: in the condition *door-open(Truck-1)*, *door-open* is the ce-type. Modality refers to whether the state/proposition is (to be made) true or false, e.g., *door-open(Truck-1)* versus ¬*door-open(Truck-1)*.

Roles

Roles of conditions and effects correspond to the arguments of the proposition being represented. A role has a role type and a role value. A role value corresponds to an argument. A role type corresponds to the argument type For example, condition *at(Truck-1,Loc-1)* has a role of type *thing* (argument 1) with value *Truck-1* and a role of type *location* (argument 2) with value *Loc-1*. Role values can be constants (frames or immediate values) or plan variables. Roles are represented as slots: the role type is the predicate and the role value is the slot value (slot filler).

Plan Variables

Plan variables represent an argument that can assume one of a set of values. They are created and bound by the planner. Plan variables are used as role values. For example, the condition *at(PVAR-T1,PVAR-L1)* has two plan variables as its arguments (role values). Each of these plan variables has a value which may be either a constant or a frame, e.g., *PVAR-T1* may be bound to *Truck-1*. Plan variables are often designed with (frame) names that start with "PVAR-".

Conditions

Conditions are associated with actions and plan validations. Conditions have attributes ce-type, modality, roles, and a condition type.

Condition types indicate how a planning system is to handle a particular condition associated with a task schema, which specifies how to decompose a particular task (e.g., goal/subgoal) during planning. The type of a condition is considered when a task is expanded (decomposed) in a generative planner such as UM NONLIN. Although task schemas are not used in CAPER, CAPER considers condition types during annotation verification (see Section 6.1.4).

Conditions types include include *precondition*, *filter-condition*, and *phantom-condition*. The first 3 types are fairly standard in planning systems (e.g., UM NONLIN, PRIAR). A precondition is a condition that must be achieved by the planner before an action can be performed.[8] A phantom condition (or phantom goal) is a precondition which already holds when a task is expanded (i.e., this condition is true in the initial state or made true by a previous action in the developing plan). For example, a precondition of going to the store is "have money". If the agent already has money, this condition is a phantom condition and need not be achieved by the planner by going to the bank, etc.

A filter condition is one that must be true before an action is executed, but the planner should not try to achieve it in the course of expanding a task. For example, a filter condition of a plan for walking to the store when it's raining (using an umbrella, etc.) is "must be raining". The planner obviously should not try to achieve this condition by making it rain. This condition must hold in initial state or made to hold by some previous action in the plan. Sometimes the latter is impossible, i.e., when no action can contribute a particular condition as an effect. An example of this is the condition *in-city(Airport-1,City-1)*, Airport-1 is located in City-1, which cannot be achieved at all by the planner: it must already hold in the initial situation. During plan mapping, CAPER makes use of knowledge about which conditions can be established and which cannot (see Section 5.3).

Goals

Goals are associated with cases and plans. Goals are conditions to be achieved by the plan for a case. Figure 3.10 shows a semantic network representation of a goal *Make-Deliv-1-Goal-1*, the sole goal for case *Case-Make-Deliveries-1* (Figure 3.7). This goal is to deliver package *Pkg-1.1* to customer location *City1-CL2*. It corresponds to the proposition *located-at(PVAR-P1,PVAR-L12B)*, where *PVAR-P1* and *PVAR-L12B* are plan variables that have been bound to package *Pkg-1.1* and customer location *City1-CL2* , respectively. The semantic net shown in the figure specifies the following:

- *Make-Deliv-1-Goal-1* is an individual goal of ce-type *CE-Located-At*, i.e., a goal of making something located at somewhere: package *Pkg-1.1* should be located at customer location *City1-CL2*.
- *Make-Deliv-1-Goal-1* has modality = True.
- *Make-Deliv-1-Goal-1* has roles of type *thing* and *location* with role values *PVAR-P1* and *PVAR-L12B*, respectively.

---

[8] One source of potential confusion is that in the A.I. planning literature, the term "precondition" might refer to a condition of any type (e.g., such as a filter condition) that must hold before an action can be executed. In this paper, "precondition" is used to mean a condition that must be achieved by the planner.

- Plan variables *PVAR-P1* and *PVAR-L12B* are bound to regular package *Pkg-1.1* and customer location *City1-CL2*, respectively.

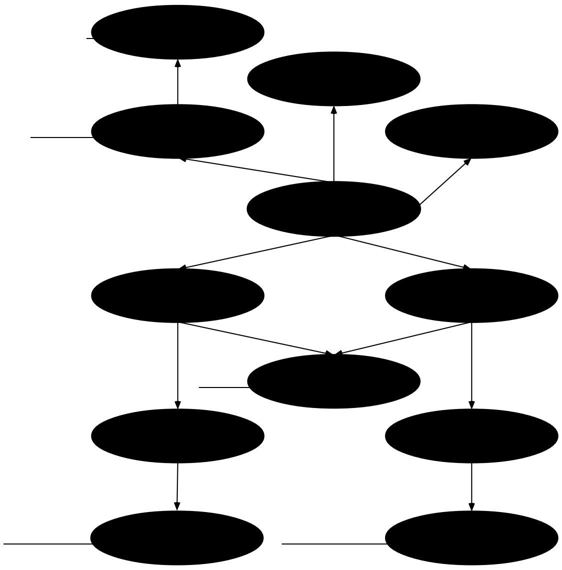

Figure 3.10: A part of the net representing the goal Case-Make-Deliv-1-Goal-1.

Effects

Effects are associated with actions. Effects have attributes ce-type, modality, and roles. These attributes are the same as those for conditions.

3.4.5 E-Actions

E-Actions (episodic actions) are a kind of episodic knowledge. They are actions that have been wholly or partially instantiated. In CAPER an e-action consists of an action type, action roles, action condition(s), action effects(s), and plan validations. A complete listing of the e-actions for plan *Plan-Located-At-1.3* is given in Appendix C.2. Plan validations are described in Section 3.4.6.

The representation of actions in terms of conditions and effects is fairly common in A.I. planning systems and is known as the STRIPS representation (Fikes and Nilsson, 1971). In this type of representation, conditions and effects are represented as a sets of first-order formulae. The STRIPS assumption, a way of handling the frame problem, states that anything in the world not explicitly mentioned in some effect formula is unchanged by the action's execution. For example, if the proposition *color(Sky,Blue)* is true before action *a* is executed, it will be true after action a is executed, unless action a has an explicitly stated effect which deletes the proposition *color(Sky,Blue)*.

An action type is a category that the e-action is an instance of. For example the individual e-action *P-Open-Door-1.1* is an instance of *P-Open-Door*. The latter is one of the domain planning concepts that has been defined.

Action Roles

Action roles represent an action's parameters, e.g., the agent, object, instrument, etc. A role has a role type and a role value. For example, e-action *P-Open-Door-1.1* has a role of type *vehicle* with a value of *Truck-1*. Role values can be constants (frames or immediate values) or plan variables (see Section 3.4.4). Roles are represented as slots: the role type is the predicate and the role value is the slot value (slot filler).

Figure 3.11 shows a semantic network representation of a portion of a particular e-action, *P-Load-Pkg-1.1*, that specifies the following:

- *P-Load-Pkg-1.1* is a individual (primitive) e-action which is an instance of *P-Load-Pkg*, an action type.
- *P-Load-Pkg-1.1* is an action of e-plan *Plan-Pkg-Loaded-1.1* (shown in detail in Figure 3.9).
- *P-Load-Pkg-1.1* has various plan validations (shown in detail in Figure 3.12). (Note only some plan validations for this action are shown.)
- In *P-Load-Pkg-1.1*, package Pkg-1.1 is loaded into truck Truck-1.
- *P-Load-Pkg-1.1* has roles of type *package* and *vehicle* with role values of plan variables *PVAR-P1* and *PVAR-T1*, respectively.
- Plan variables *PVAR-P1* and *PVAR-T1* are bound to regular package *Pkg-1.1* and truck *Truck-1*, respectively.

Recall that actions are the leaf nodes of the HTN (Section 3.4.3). Actions are connected to plans by *actionOf* links (the inverse of *Action* links). Action membership in plans is transitive and uses PARKA's transitive inference mechanism (Section 3.1.2). For example, if action a is directly an action of plan p_1 and plan p_1 is a subplan of plan p_2, then action a is indirectly an action of plan p_2.

3.4.6 Plan Validations

Plan Validations represent dependencies among the components of a plan. They are based on the plan validations used in PRIAR (Kambhampati and Hendler, 1992) and consist of a condition, contributor, and consumer. E-plans and e-actions have (are annotated with) plan validations. The set of plan validations for a plan constitutes the plan's validation

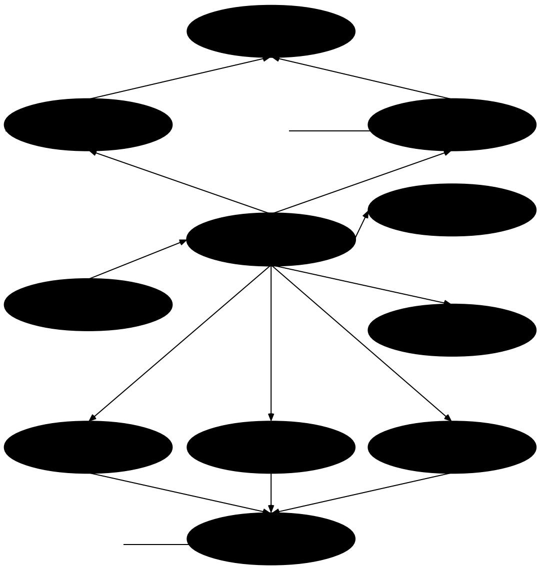

Figure 3.11: A part of the net representing the e-Action P-Load-Pkg-1.1.

structure. The plan validation structure is used during plan mapping (Section 5.3), checking (Section 6.1), and adaptation (Section 6.2). A complete listing of the validations for plan *Plan-Located-At-1.3* is given in Appendix C.2.

A plan validation captures information about a kind of plan dependency, i.e., when one HTN node n_d, the consumer, has a condition that is satisfied by an effect of another node n_s, the contributor. A node is an HTN node: an e-action, an e-plan, or the special nodes n_i and n_g, which are dummy nodes representing the initial and goal states of a plan. The consumer node precedes the contributor node in the plan. Conditions are described in Section 3.4.4.

Each HTN node n (an e-action, e-plan, n_i, or n_g) has three sets of plan validations, each of which may be empty. These sets are e-conditions, e-preconditions, and p-conditions. Each validation may be a member of one (or none) of these sets per node, e.g., validation V1 may be an e-condition of node N1, an e-precondition of node N2, an e-precondition of node N3, a p-condition of node N4, etc. The contents of these sets are determined by the reduction of the node n: $R(n) = n \cup \text{descendants}(n)$, where descendants($n$) are the children of n in the HTN.

E-Conditions

E-conditions (external effect conditions) are externally useful validations: e-conditions(n) are those effects of nodes in $R(n)$ that are used to satisfy conditions of nodes not in $R(n)$.

E-Preconditions

E-preconditions (external preconditions) are externally established validations: e-preconditions(n) are those conditions of nodes in $R(n)$ that are satisfied by effects of nodes not in $R(n)$.

P-Conditions

P-conditions (persistence conditions) are those conditions contributed and consumed by nodes not in $R(n)$ which must persist through (i.e., not be clobbered by) nodes in $R(n)$.

Figure 3.12 shows a semantic network representation of a portion of a particular plan validation, *Plan-Vdtn-1.7*, that specifies the following:

- *Plan-Vdtn-1.7* is a individual plan validation.
- *Plan-Vdtn-1.7* has condition *Plan-Vdtn-1.7-Cond*, stating that package *Pkg-1.1* is to be located inside truck *Truck-1*.
- *Plan-Vdtn-1.7-Cond* is contributed by e-action *P-Load-Pkg-1.1* (shown in detail in Figure 3.11) as one of its effects and consumed by e-action *P-Close-Door-1.1*. The condition is a precondition of *P-Close-Door-1.1*. Both actions are in the subplan *Plan-Pkg-Loaded-1.1* (shown in detail in Figure 3.9).
- *Plan-Vdtn-1.7* is an e-condition validation of e-action *P-Load-Pkg-1.1* and a p-condition of e-action *P-Load-Pkg-1.1*.

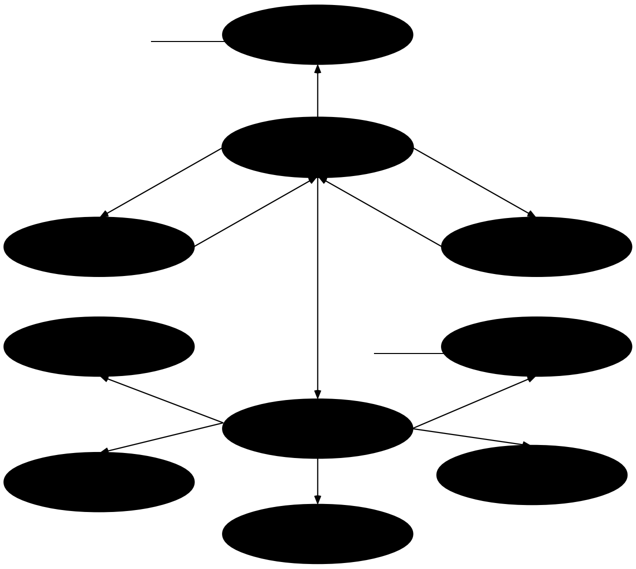

Figure 3.12: A part of the net representing the plan validation Plan-Vdtn-1.7.

3.5 Seeding the Casebase

In order to solve problems in a new domain, CAPER requires an initial or "seed" casebase. In most CBR systems, the initial casebase must be built by manual methods. Cases may need to be acquired from knowledge in other forms (such as from the memory of an expert, for example). Even if there is existing knowledge in the form of cases (such as in a medical and legal domains, for example), some knowledge engineering process must be used to put the cases online in a form usable by the CBR system. For CBR systems that use an indexed casebase, the indexing scheme and the indices of individual cases must be determined. Since the knowledge acquisition and knowledge engineering processes are typically expensive, many CBR systems start with a small casebase (i.e., in the tens of cases).

CAPER currently uses UM NONLIN (Ghosh *et al.*, 1992), a Common Lisp version of NONLIN (Tate, 1977), to generate its seed casebase. UM NONLIN is run on randomly-generated problems for the UM TRANSLOG (Andrews *et al.*, 1995). The correct plans output from UM NONLIN are automatically stored as cases in the initial casebase.

Figure 3.13 shows the modules and data Casebase Seeder to generate the initial casebase. These components, described below, are configured for a particular domain (e.g., UM TRANSLOG) by specifying custom CLOS classes and methods for domain objects (e.g., cities, trucks, packages). CAPER's domains are described in Appendix A.

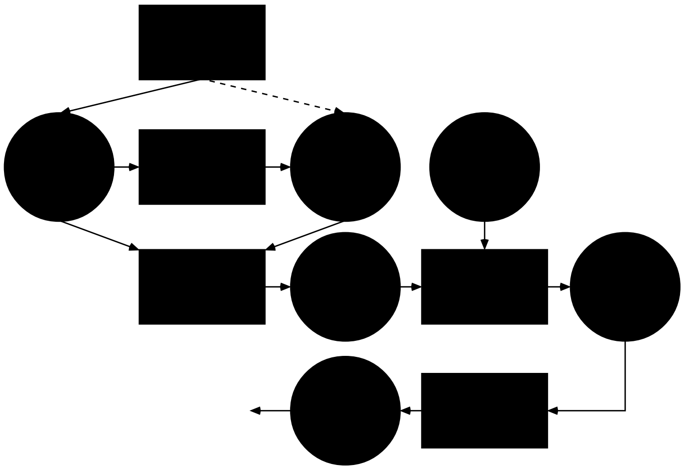

Figure 3.13: Case Seeder components and process. (Boxes are modules. Circles are data).

The Planner component is currently the UM NONLIN planner. Another planning system could be used by supplying the appropriate Planner Interface and Planner modules and Domain Operators (schema definitions).

A future way to exploit the UM NONLIN/CAPER connection would be to have UM NONLIN (or another generative planner) spend its "idle" time generating plans for a range of potential planning problems it may encounter. These plans would then be available and in the event that one of these planning problems is encountered, CAPER could merely retrieve the stored plan. This of course depends on having fast retrieval procedures for very large casebases. Humans frequently engage in this kind of contingency planning.

3.5.1 Problem Specifier

A planning problem is defined by a user via the Problem Specifier. The user specifies a set of statements in a macro language. These statements in turn define a set of entities and goals. These entities are represented in a planner-independent manner (via CLOS objects). The Problem Specifier produces a world definition and a problem definition, which can be used as input to the Case Seeder's Planner (e.g., UM NONLIN) or as input to CAPER (for a problem that CAPER is to solve).

World Definitions

A world definition defines the static facts of the world. For example, in the UM TRANSLOG domain (Section A.1), a world definition defines what cities there are, what customer locations there are and which cities they are located in, what transport centers are where, etc. A world definition is specified by the user via the Problem Specifier.

Problem Definitions

A problem definition defines the facts of the world that constitute the initial situation of a target problem. It also defines the goal(s) of the target problem. In the UM TRANSLOG domain, a problem definition specifies the initial location and availability of vehicles, type and initial location of packages, and goal destinations for packages. A problem definition can be specified manually by a user or automatically generated by the Problem Generator. A world definition corresponding to the world in which the problem is to be solved must be provided before a problem definition can be created.

3.5.2 Problem Generator

The Problem Generator can be used to generate random problem definitions for a user-specified world definition. A user supplies the world definition, specified via the Problem Specifier, and several parameters. The Problem Generator produces the requested number of Problem Definitions.

3.5.3 Planner Interface

Once a world definition and problem definition have been created using the Problem Specifier/Problem Generator, they can be given to a planner as input. The Planner Interface is a set of functions for a given planning system that takes these domain-dependent definitions and converts them into input in an appropriate format for the particular planning system. Currently a planner interface exists for CAPER (see Section 4.4) and UM NONLIN.

The planner (e.g., UM NONLIN) is given this input and a set of planning operators (schemas) for the domain. The planner runs and produces output. Output from UM NONLIN includes the following:

1. the planning problem — domain problem and world definitions
2. the plan — a list of actions (and role fillers) generated by the planner
3. the plan variables — variables and bindings used in generating the plan
4. the plan's HTN structure — the task decompositions used to create the plan
5. the plan's validation structure — the validations for the plan

UM NONLIN was modified to output the plan variables, HTN structure, and validation structure[9]

3.5.4 Memory Generator

The domain-independent Memory Generator takes the planner output (file) from UM NONLIN and creates a case definition (file) for CAPER. A case definition file is a Lisp source file containing statements that, when loaded, create the appropriate PARKA frames for the case. Code is produced by the Memory Generator that defines the e-actions, e-plans, plan validations, conditions, goals, and the case frames for the particular plan solved by the planner.

3.6 Chapter Summary

The following were discussed in this chapter.

- The parallel, frame-based knowledge representation system PARKA is used to implement CAPER's persistent KB, a semantic network. PARKA provides an expressive language, a variety of inference methods, and fast query-processing mechanisms. The latter include the PARKA Structure Matcher, which processes structure queries used extensively by CAPER.
- CAPER's knowledge representation scheme provides a way to express a variety of knowledge about concepts, individuals, and episodes, taking advantage of PARKA's inheritance capabilities. As discussed in Section 2.2.1, graph-structured case representations provide greater expressivity and support for determining structural similarity between cases.

---

[9] The validation structure was created using PRIAR's annotation routines designed and initially implemented by Rao Kambhampati and implemented for UM NONLIN by Brian Kettler.

- Episodic knowledge represented includes cases, e-plans, e-actions, plan validations, goals, conditions, effects, roles, and plan variables. Within a plan, the hierarchy of subplans and actions is called the HTN.

- CAPER's Casebase Seeder component provides a way to automatically seed an arbitrarily large casebase. Plans produced by UM NONLIN are annotated with plan validations and are stored as cases in the casebase. As discussed in Section 2.1, a larger casebase provides more expertise for a CBP system to draw upon.

Chapter 4

Retrieval in CAPER

4.1 Overview

CBP systems need mechanisms for effective and efficient case retrieval, as was argued in Section 2.3. This chapter describes CAPER's data structures and algorithms for case (plan) retrieval that comprise its Plan Retriever module, designed to provide fast, frequent, and flexible retrieval.

Fast retrieval is provided by the underlying parallel mechanisms of PARKA (Section 3.1). As will be shown in Section 4.7, the PARKA Structure Matcher supports fast, scalable matching of a probe graph to CAPER's KB, a single large semantic network described in the previous chapter. This subgraph-matching approach is possible due to the graph-structured case representation scheme CAPER uses (Section 2.2.1).

Because the underlying retrieval process is fast, memory does *not* have to be pre-indexed (Section 2.3.2). This provides additional flexibility at case retrieval time. CAPER forms retrieval probes dynamically using aspects of the target problem (or subproblem) and domain knowledge, if available. The probe can contain any nodes and arcs (i.e., any frames defined in the KB). Thus CAPER can retrieve an entire case plan, or part of a plan, via any of its constituent frames. Probes can be of varying specificity. If a retrieval probe fails, CAPER can modify it dynamically by adding, deleting, or replacing nodes and arcs.

With fast retrieval, CAPER can afford to go to memory often. It does this when a probe fails, and it then has to issue one or more modified probes. The retrieval process can be recursively invoked by the Plan Modifier to retrieve a replacement or patch plan that achieves a target (sub)goal.

Before the discussion of the data and modules involved in plan retrieval, a simple example will be described in the following section.

4.2 An Example

The section presents an example of plan retrieval from the UM TRANSLOG domain (Appendix A.1). The sections that follow discuss the plan retrieval mechanisms in more detail.

The target planning problem to solve is *Case-Make-Deliveries-3*, or Case-3 for short. This problem will be used as a running example through this chapter and those that follow. In this problem, a non-hazardous, non-valuable, liquid package/cargo (e.g., water) is to be delivered from *City1-CL1* to *City1-CL2*, two customer locations in the city *City-1*. This fairly simple problem is solved by CAPER.

Figure 4.1 show a schematic view of the UM TRANSLOG world *World-1* in which Case-3 occurs. Three cities are shown along with the airports, train stations, and customer location in each.[1] The initial situation in Case-3 is:

- A non-hazardous, non-valuable, liquid package *Pkg-3.1* is at customer location *City1-CL1*.
- An empty, available, regular-type truck *Truck-1* is located at customer location *City1-CL1*.
- An empty, available, tanker-type truck *Truck-1* is located at customer location *City1-CL1*.

The world and planning problem are specified to CAPER via a world definition (Section 3.5.1) and a problem definition (Section 3.5.1), respectively. A target problem is represented in CAPER's working data by a **problem description** (Section 4.4.2).

The plan for Case-3 will be fairly simple. Liquid package *Pkg-3.1* has to be carried in a compatible truck, the tanker truck *Truck-3*. *Truck-3* will need to load *Pkg-3.1* and deliver it to *City1-CL2*. As both *City1-CL1* and *City1-CL2* are locations in the same city, *City1*, there is assumed to be a road connection between them.

For the simple example, we will assume a small test KB is being used here that contains two cases:

- *Case-Make-Deliveries-1* (Case-1) contains a plan, *Plan-Located-At-1.3* (Plan-1), for delivering a regular-type package, *Pkg-1.1*, using a regular-type truck, *Truck-1*, from customer location *City1-CL1* to *City1-CL2*.
- *Case-Make-Deliveries-2* (*Case-2*) contains a plan, *Plan-Located-At-2.4* (Plan-2), for delivering a liquid-type package, *Pkg-2.1*, using a tanker truck, *Truck-3*, from customer location *City1-CL2* to *City1-CL1*.

The plans for these two cases were generated using UM NONLIN and the Casebase Seeder (Section 3.5).

Some new frames for the KB are created by the Problem Analyzer to represent the initial state and goals of Case-3. These are shown in Figure 4.2. The representation scheme for cases in the KB was described in the previous chapter.

---

[1] As discussed in Appendix A.1, the precise geographical locations of cities are not modeled.

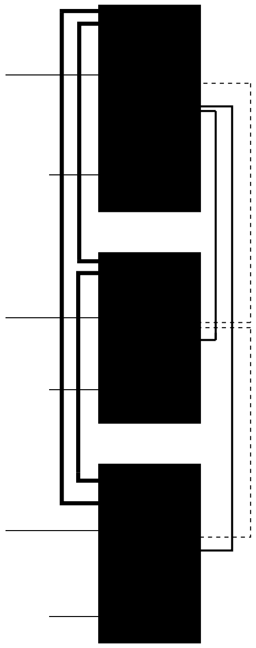

Figure 4.1: Schematic Diagram of locations and intercity routes of World-1. (Intercity road routes exist between all locations within a city. Intercity air and rail routes are omitted for clarity.)

When retrieving a plan, the Plan Retriever first tries to find a source plan in the KB that solves a problem as closely matched to the target problem as possible. It does this by issuing a query in the form of a **probe graph** to the PARKA Structure Matcher (Section 3.1.4). If the probe matches one or more plans, a similarity metric is applied to rank the candidate source plans retrieved. The highest ranking of which is chosen for further processing. If the probe fails, it is modified and reissued.

An initial retrieval probe graph is created that specifies the aspects of the target case to be matched to source cases. There is great flexibility in this because CAPER's memory is not preindexed, and hence CAPER is not limited to a set of predefined indices for accessing cases. A retrieval probe is actually represented in CAPER's working data by a **retrieval description** (Section 4.6.1), a data structure specifying the kind of information desired from memory.

In the UM TRANSLOG domain, the initial retrieval description contains the goal of the target plan with the specific origin, destination, and package type. The package has been variabilized because a particular package does not occur in other cases. The decision to use these aspects of UM TRANSLOG target problems is a type of domain knowledge.

The probe graph corresponding to the initial retrieval description for Case-3, *RDESC-1*, is shown in Figure 4.3. It has query variables ?PLAN, ?GOAL, and ?ISIT which can match any

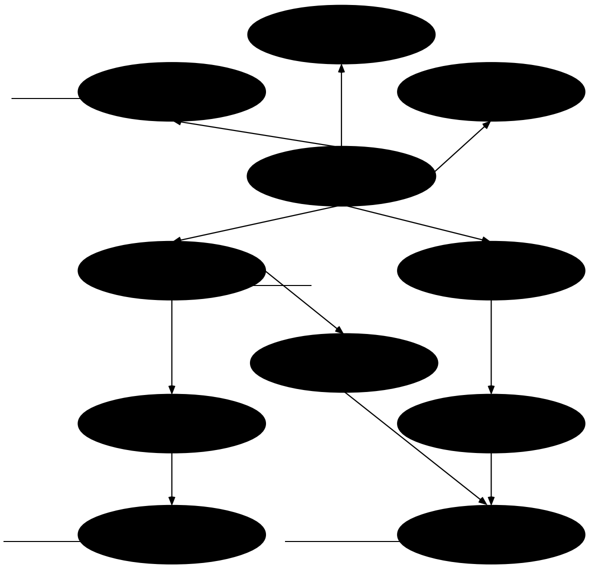

Figure 4.2: The subgraph of Case-3's single goal, Make-Deliv-3-Goal-1. (As before, instance node labels are underlined, category node labels are not.)

e-plan, plan goal, and initial situation, respectively. Only plans with a package destination of *City1-CL2* will be matched by this probe, i.e., query variable ?DEST-PVAR must match a plan variable (Section 3.4.4) whose value is *City1-CL2*. Thus the package destination in the probe is a *constant*. The package origin is also a constant in the probe, because any package matched must have its location slot filled with *City1-CL1* in the situation bound to query variable ?ISIT (*location* is a situated slot, see Section 3.3). Query variable ?PKG-PVAR can match any plan variable whose value is of type *Liquid-Package*. Thus the package in the probe has been *variabilized*.

The PARKA Structure Matcher tries to match the probe to the KB, fails, and returns NIL. Because the PARKA Structure Matcher must find an exact match (see Section 3.1.4), the probe does *not* match the subgraph of the KB representing Case-1 that is shown in Figure 4.4. Case-1 is not matched because the type of the package delivered there is a *regular* package, not a *liquid* package as specified by the probe. Case-2 is not matched because its origin and destination locations do not match those of the probe.

Given a failing retrieval description, the Retriever must modify its retrieval description so that it can find one or more source plans to work with. There are typically several possible modifications that can be applied to a retrieval description. Each results in a different retrieval probe. A probe can be generalized by replacing a constant node with a query variable; replacing a constant node or predicate (link) with a supercategory; removing nodes and links, etc. A probe can be specialized by doing the reverse: i.e., replacing a node with a subcategory, etc.

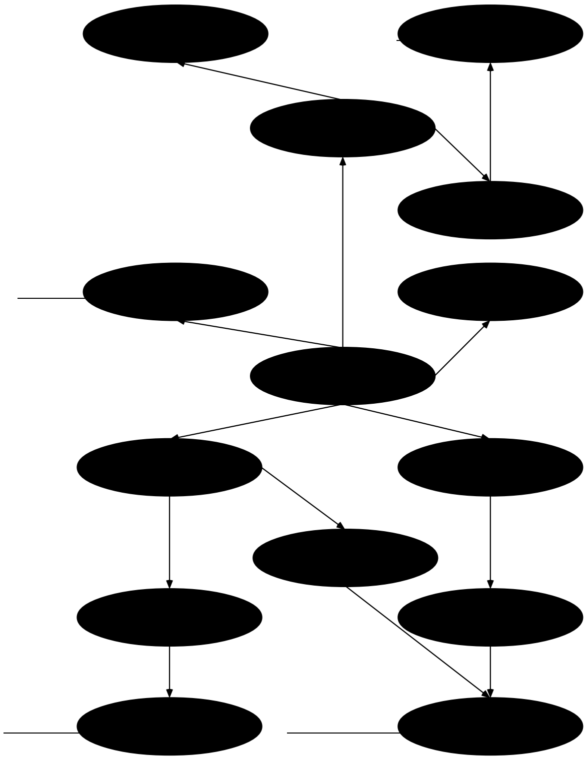

Figure 4.3: The probe graph for RDESC-1: find plans for delivering a liquid-type package from City1-CL1 to City1-CL2. (Nodes with labels prefaced by a "?" are query variables.)

The failing retrieval description *RDESC-1* (probe) can be generalized as follows:

1. variabilize the origin (replace *City1-CL1* with a variable)
2. variabilize the destination (replace *City1-CL2* with a variable)
3. generalize the package type (replace *Liquid-Package* with a supercategory)
4. generalize the goal (replace *CE-Located-At* with a supercategory)

The above modifications are conceptualized as retrieval description operators in CAPER. The decision about which retrieval description operator to apply is treated as a search task, described in Section 4.6.3. Section 4.6.4 give details on how modifications are applied, etc. for this particular example.

The Retriever has to decide (a) which one or more of the applicable modifications (1)–(4) should be applied and (b) how to apply it to the retrieval description. The latter involves selecting the particular supercategory to use for modifications (1) and (2), which is more complicated if there are multiple to choose from (see Section 4.6.5).

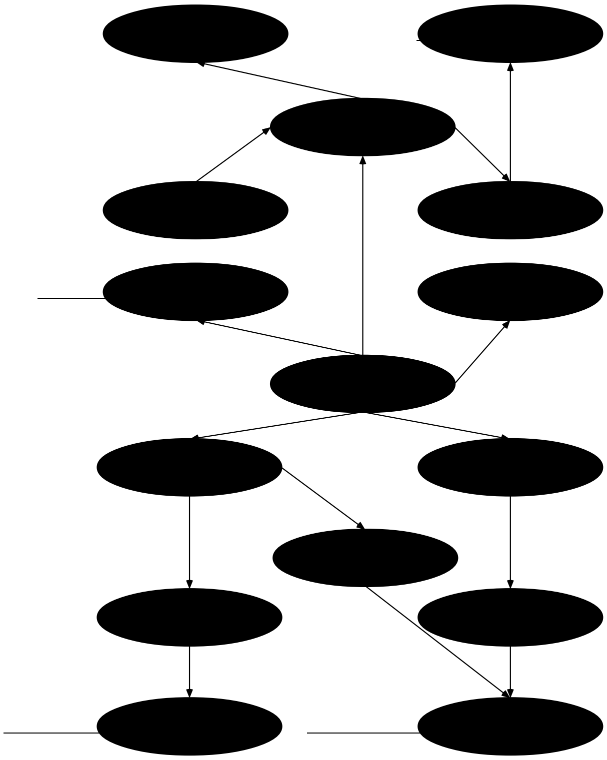

Figure 4.4: Part of the semantic net representing Case-Make-Deliveries-1

The modification chosen for *RDESC-1* is (3), generalize the package type. It is chosen using the domain-specific heuristic that it is better to match the origin and destination locations than to match the package type, if both cannot be matched by some source plan. In other words, less modification is required if the the source plan differs in the package type rather than in the origin or destination.

Typically a probe is generalized until it returns plans or until the probe (or plans retrieved) exceeds some domain-specific metric of imprecision. If package type were to be generalized too far up the abstraction hierarchy to *PhysicalObject*, for example, precision would suffer (e.g., plans for transporting people or other kinds of "physical objects" might be matched). The more plans returned by the PARKA Structure Matcher, the more work that must be done to rank the candidate source plans.

The package type in *RDESC-1* in generalized from *Liquid-Package* to its supercategory, *Package*. The probe corresponding to resulting retrieval description, *RDESC-2*, is shown in Figure 4.5. Note that two changes have been made from the probe in Figure 4.3. The node for *Regular-Package* has been replaced with a node for *Package*. The *instanceOf* link connected to this node has been replaced by its transitive form, *instanceOf**.

The new probe is issued and succeeds. Two sources plans are matched for *Case-Make-*

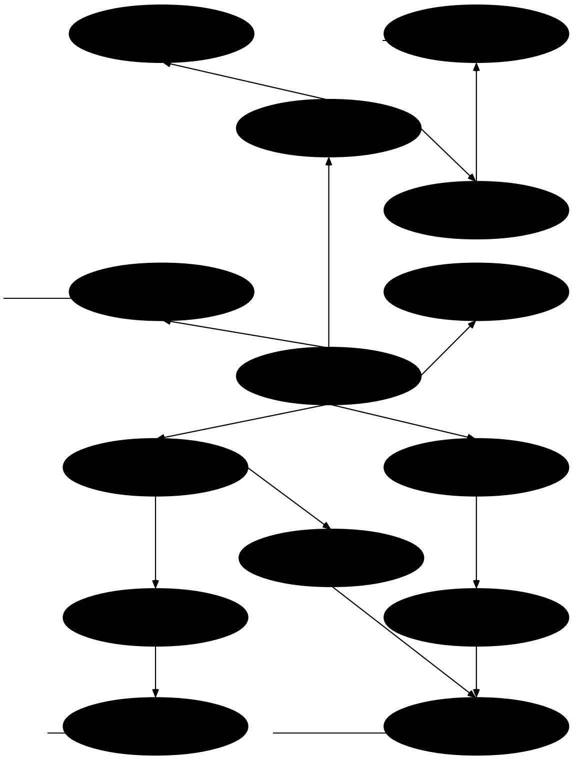

Figure 4.5: The probe graph for RDESC-2: find plans for delivering *any* package from City1-CL1 to City1-CL2.

Deliveries-1: *Plan-Located-At-1.3* and *Plan-Located-At-1.2*. Both plans achieve the goal of delivering a *Regular-Package* from *City1-CL1* to *City1-CL2*. The latter plan is selected because it is a subplan of the former plan. More specific plans are preferred over more general plans. CAPER applies a domain-specific similarity metric that ranks candidate source plans (i.e., those matched by the probe).

Had this probe failed, CAPER would have to form a new retrieval description, either by applying one of the above modifications to *RDESC-1* not tried — (1), (2), or (4) — or by applying a modification (1)–(4) to *RDESC-2*.

The highest ranking plan, *Plan-Located-At-1.2*, is chosen for further processing. It is shown in Figure 3.8 (see Appendix C.2 for details). The entities in this plan (e.g., *Pkg-1.1*) will need to be mapped to ones in the target problem. This is done by the Plan Mapper as will be described in the following chapter. Because the source plan's package is a regular package (e.g., a box) and the target case's package is a liquid package, some adaptation of the source plan will be required. This is done by the Plan Modifier as will be described in Chapter 6. The details of the retrieval process are described in the remainder of this chapter.

4.3 The Details

The remaining sections in this chapter describe at an algorithmic level the modules and data used during plan retrieval.

In brief, plan retrieval involves the following top-level modules, shown in Figure 4.6:

1. The Problem Analyzer (Section 4.4) converts a world and problem definition into a set of new KB frames (and assertions) and a problem description, which contains a set of case features.

2. The Plan Retriever (Section 4.5) takes a problem description as input. It formulates a retrieval plan, which contains a series of retrieval descriptions.

3. The Retriever (Section 4.6) takes a retrieval description, converts it to a retrieval probe(s), and passes it to the PARKA Structure Matcher for "execution". A successful retrieval description results in a set of candidate source plans and their descriptors. If unsuccessful, a retrieval description is modified for re-execution by finding an applicable retrieval description operator. The Retriever is thus searching a space of retrieval descriptions to find one that matches one or more source plans.

4. The Plan Selecter (Section 4.5.1) takes a set of candidate source plans, ranks them using a similarity metric, and chooses the best for further processing.

As mentioned in the previous chapter, CAPER uses working data, temporary information associated with persistent knowledge. Figure 4.7 shows the various kinds of working data used during plan retrieval and their relationships to each other. These kinds will be described in the remainder of this chapter along with the various modules that use them.

4.4 Problem Analyzer

The Problem Analyzer module is invoked by CAPER's Main module. Input to the Problem Analyzer is a planner-independent world definition (Section 3.5.1) and problem definition(Section 3.5.1), which describe the target problem. The Problem Analyzer does the following:

1. creates a new problem description (Section 4.4.2) for a target problem containing relevant case features

2. creates new frames for the target goal(s)

3. updates the frames for the target problem's initial state

Domain-specific methods may be supplied for generating the initial frames, initial target features, and initial retrieval plan (Section 4.6.6). For example, the initial frames for Case-3 were shown in Figure 4.2.

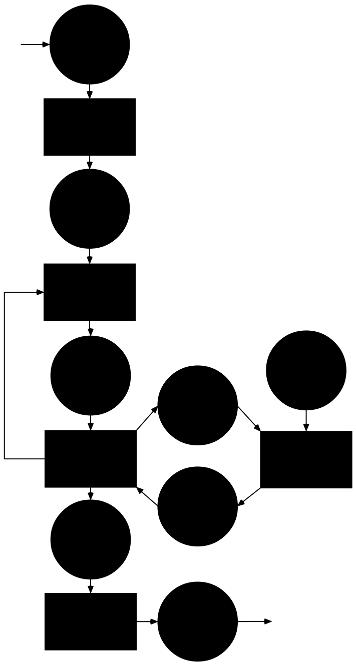

Figure 4.6: Plan Retrieval modules and the flow of control.

4.4.1 Case Features

The facts that comprise a previous case are represented in the KB as a part of the semantic network (implemented via PARKA frames and assertions). In CAPER's working data, some facts of the target case and source cases are represented by case features. A case feature is a (temporary) data structure that corresponds to one or more case facts. Thus a case feature corresponds to a subgraph of a case's subgraph in the semantic network. Case features provide an additional way of representing parts of cases in CAPER. They are used to modify a retrieval description, to apply similarity metrics, and to establish correspondences between a target and source case,

For example, *Feature-3.1*, is an alternate representation for the goal of Case-3, whose graph-structured semantic net representation was shown in Figure 4.2. This feature is an instance of feature class *UMT-GOAL-TRANSPORT-PKG* and has six feature facets, described below. A feature is an instance of a feature class which has been defined by a domain engineer. Various methods are associated with each feature class. Details about the contents of case features are given in Appendix B.1.1.

Feature facets are components of case features. A feature facet describes an aspect of what the case feature represents. Facets are either constants (representing known entities) or variables (representing unknown entities). Variable facets have associated variable types. A facet is an instance of a facet class which has been defined by a domain engineer. Various

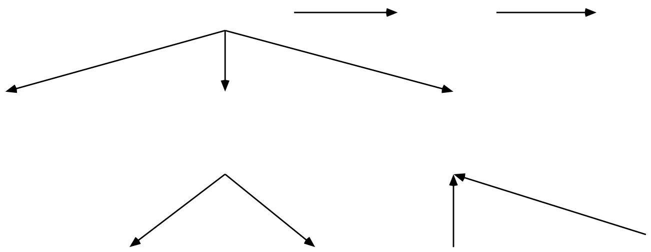

Figure 4.7: Kinds of working data used during plan retrieval and their interrelationships.

methods are associated with each facet class. Details about the contents of feature facets are given in Appendix B.1.2.

For example, *Feature-3.1*, representing the target goal, has six facets:

- **Package** with initial value *Pkg-3.1* (Class *UMT-FCT-PKG*)
- **Origin** with initial value *City1-CL1* (Class *UMT-FCT-LOC-GEO*)
- **Destination** with initial value *City1-CL2* (Class *UMT-FCT-LOC-GEO*)
- **Plan** with initial value *Plan-3*, an empty plan (Class *UMT-FCT-PLAN*)
- **Goal** with initial value *Case-Make-Deliv-3-Goal-1* (Class *UMT-FCT-GOAL*)
- **Situation** with initial value *Case-Make-Deliv-3-InitSit*, the initial situation of Case-3 (Class *UMT-FCT-ISIT*)

The initial case features are generated by invoking a domain-specific procedure which instantiates one or more features from feature (and facet) classes that have been defined by a domain-engineer. Methods associated with feature and facet classes specify how the facets of newly instantiated feature are to be filled. Feature facets are initialized to values from the target planning problem's initial situation.

While some of the plan retrieval algorithms make use of case features, it is important to note that the case representation is still graph-structured (Section 2.2.1) and PARKA Structure Matcher probe graphs are used to match cases in the KB. In essence, a case feature contains a view (reification) of a part of the semantic network to which metaknowledge and temporary bookkeeping information can be attached.

4.4.2 Problem Description

A collection of case features describing a target problem is stored in a problem description. A problem description is a data structure that holds temporary information about a target planning problem. In addition to a set of case features, a problem description stores descriptors of source plans retrieved (Section 4.5.2), target plans (including the top-level "root" target plan and its HTN), and subproblems. Details about the contents of problem descriptions and an example are given in Appendix B.1.3.

4.5 Plan Retriever

The Plan Retriever module is CAPER's top-level module for retrieving source plans from the casebase. It is invoked by CAPER's Main module to retrieve an initial source plan for the top-level ("root") target planning problem. It is also invoked to retrieve subsequent source plans for subproblems of the root target problem.

The Plan Retriever takes a target problem (or subproblem) description, retrieves source plans from the KB, and returns a chosen source plan. Candidate source plans are retrieved from the KB by the low-level Retriever module, which invokes the PARKA Structure Matcher. These are ranked and returned. The highest ranking source plan is chosen for further processing by the Plan Mapper, etc.

The top-level Plan Retriever procedure, **retrieve-plan**, does the following:

1. The **Retriever** (Section 4.6) is called with an initial retrieval description (Section 4.6.1) or a retrieval plan (Section 4.6.6) consisting of one or more retrieval descriptions. The Retriever returns a set of sets of bindings for query variables in a successful probe graph. Query variables are bound to entities in the KB (e.g., source plans, etc.). The retrieval description with the successful probe graph is returned. This retrieval description is either the initial retrieval description passed to the Retriever or a modified version of it.

2. If no variable bindings were returned, the Plan Retriever procedure terminates and signals failure.

3. If variable bindings were returned, a source plan descriptor (Section 4.5.2) is created for each source plan in the group of **candidate source plans**, those plans in the KB matched by the successful probe graph.[2] Case features are instantiated for each source plan and stored in its source plan descriptor. The mappings between source features and target features are also stored in the source plan descriptor. The mappings are generated from the variable bindings.

---

[2] It is possible for a particular source plan to be in this group more than once if multiple, differing subgraphs of that plan are matched by the probe.

4. The candidate source plans are then ranked using a domain-specific similarity metric. For each source plan, this metric weights each match (or lack of a match) for a target case feature (from the target problem description) to a feature in the source plan.

5. The highest-ranking source plan (and its source plan descriptor) are returned by the Plan Retriever.

4.5.1 Similarity Metrics

The similarity metric is applied by the Plan Selecter component. The similarity metric used to rank candidate sources is a domain-specific heuristic. In the UM TRANSLOG domain, for example, each case feature has a set of facet weights. These weights are associated with a case feature's class. The facet weights for *Feature-3,1* are: Package (1), Origin (10), and Destination (100). This means that given *Feature-3.1* and its corresponding feature f_i in source plan i, a match in values for the destination facet is preferred (i.e., has higher weight) than a match in origin or package.

In general, assume there are m candidate sources (e.g., plans): $s_1, s_2, \ldots, s_m$ whose similarity to the target problem b is to be computed. For each target feature in f_t in b, a **feature score** is computed for each $s_j, j = 1$ to m as follows.

1. Compare each source feature f_j with f_t, where f_j is the feature of source s_j matched to f_t by the Retriever as follows.

2. For each facet c_i in f_t, compare v_{t_i}, the value of c_i in f_t, with v_{j_i}, the value of c_i in f_j as follows.

3. Compute the most specific common ancestor a_{j_i} is computed for v_{t_i} and v_{j_i}. Category g_1 is more specific than category g_2 if g_1 is an immediate or indirect subcategory of g_2.

4. Assume that a_{j_k} is the most specific of the values a_j. Then source s_k is awarded $w_i \times v$ points to its overall score (initially 0), where w_i is the weight for facet c_i and v is the weight for f_t. Feature weights are currently set to 1 in UM TRANSLOG (i.e., all feature scores are weighted the same).

The feature score is computed for each feature over all the candidate sources. Sources are then ranked in descending order by score.

4.5.2 Plan Descriptions

A **plan descriptor** contains (temporary) information about a source or target plan. A **source plan descriptor** is a plan descriptor that describes a source plan retrieved by the Plan Retriever which contains case features instantiated to describe the source case, the retrieval plan used to retrieve the source plan, the ranking of the source plan, and mapping information to be used by the Plan Mapper. Details about the contents of source plan descriptors and an example are given in Appendix B.1.7.

4.6 Retriever

CAPER's Retriever module handles "low-level" retrieval of items from the KB via the PARKA Structure Matcher (Section 3.1.4). It currently is only used to retrieve plans (and subplans) from the KB, but could also be used to retrieve other kinds of knowledge from the KB (see Section 9.1.2).

From the Plan Retriever, the Retriever gets as input a retrieval description (Section 4.6.1), a set of case features describing the target case. It attempts find source plans in the KB with similar features. A probe graph corresponding to the retrieval description is matched to the KB by the PARKA Structure Matcher, which returns a set of query variable bindings if successful.

If a retrieval description is unsuccessful (i.e., no source is matched in the KB), the Retriever modifies the retrieval description to create a new retrieval description which is then processed. A retrieval description is modified by applying an applicable retrieval description operator (Section 4.6.2). This process repeats until some retrieval description is successful or no more retrieval descriptions can be generated (i.e., there are no more untried retrieval descriptions nor applicable retrieval description operators). Thus, the Retriever searches a space of retrieval descriptions until one is found which results in source plan(s) in the KB being matched.

Thus there are two kinds of search going on here:

- a search of the space of retrieval descriptions by the Retrieval Description Search procedure (Section 4.6.3).
- a search of the KB by the PARKA Structure Matcher to match a probe to a plan

4.6.1 Retrieval Descriptions

A **retrieval description** is a working data structure which specifies those case features that have been (or will be) used to retrieve a relevant source case (plan) from memory.[3] An retrieval description contains case features derived from those in the problem description. It also contains the corresponding probe graph and its results (query variable bindings, etc.). Details about the contents of retrieval descriptions and an example are given in Appendix B.1.4.

For example, retrieval description *RDESC-1* has the single case feature *Feature-3.1.* Its corresponding PARKA Structure Matcher probe graph is shown in Figure 4.3. This retrieval description was used in the example in Section 4.2 to retrieve source cases from memory that contain a feature similar to *Feature-3.1*, i.e., source cases that have a package delivery goal similar to that of Case-3.

---

[3] A retrieval description could also be used to retrieve other kinds of knowledge from memory (e.g., objects matching a description), but CAPER currently does not take advantage of this capability.

4.6.2 Retrieval Description Operators

A **retrieval description operator** is a working data structure that specifies how a retrieval description can be modified to produce a new retrieval description. This is done, as will be described below, when a query corresponding to the retrieval description fails. A retrieval description r can be modified in several general ways:

- adding, deleting, or modifying (generalizing, specializing, etc.) a case feature in r;
- adding, deleting, or modifying (generalizing, specializing, etc.) a facet of a case feature in r;
- any combination of the above.

A retrieval description operator contains the changes to be made to a retrieval description or its components. In the example of Section 4.2, retrieval description operator *RDOP-1* modifies retrieval description *RDESC-1* by modifying its case feature *Feature-3.1.* This case feature is modified by generalizing the type of its variable facet representing the package to delivered from *Liquid-Package* to its supercategory *Package.* Details about the contents of retrieval description operators are given in Appendix B.1.5. In addition to domain-independent retrieval description operators such as Variabilize-Facet, domain-dependent retrieval description operators can also be defined by a domain engineer.

4.6.3 Retrieval Description Search Procedures

Procedure **rdesc-search** does graph search over the space of retrieval descriptions. The search space contains nodes which are termed **RDS nodes** in order to distinguish them from nodes in the semantic network. A retrieval description is associated with each node (state in the search space). Arcs in the search space are (instantiated) retrieval description operators.

The following is the search algorithm which terminates when a successful retrieval description is found or the search space is exhausted.

```
Procedure rdesc-search (init-rdesc : retrieval description)

variables
   n, m : RDS node
   open-list,
   closed-list : list of open (unexpanded) nodes;

begin
   n := make new node for init-rdesc;
   put n on open-list

   loop until n is a goal node or open-list is empty
```

```
      order open-list
      n := highest ranking node on open-list
      remove n from open-list
      put n on closed-list
      if not goal-node(n) then
         for each m in node-successors(n), put m on open-list
   end loop

   if a goal node n was found then
      return the successful rdesc of n,
             the sources matched by it, and
             its retrieval plan as the result
   else
      return ''failure''
   end if
end.
```

The open list is ordered by sorting its nodes according to depth in the search tree, breadth, or by score. In the UM TRANSLOG domain, node scores are used to sort the open list. These scores are based on the general type of the domain operator (variabilize. generalize, etc.) and on the case feature and feature facet class (*UMT-FCT-PKG*, *UMT-FCT-LOC-GEO*, etc.).

When a node is expanded, its successors are installed on the open list. No check is made to see if these successor nodes have been "seen" before (i.e., are currently on the open or closed lists). This check could be performed, but it would be expensive when the open and closed lists are long. The price paid for not checking is that an occasional retrieval description may be generated more than once and processed as a PARKA Structure Matcher query. This redundant processing, however, is not very costly as queries are cheap.

The following procedures are the auxiliary routines used by **rdesc-search**.

The following procedure generates the successors of a node by applying retrieval description operators to their retrieval descriptions.

```
Procedures node-successors (n : RDS node)

variables
   old-rdesc : node n's retrieval description
   applicable-rdops : set of retrieval description operators
   succs : set of RDS nodes
   m : RDS node

begin
   applicable-rdops := applicable-rdesc-rdops(old-rdesc);
   for each rdop in applicable rdops,
      create a new node m for <old-rdesc,rdop> with parent n
```

```
      compute a score for m
      add m to succcs
   end for
   return succs
end.
```

Applicable retrieval description operators are found by examining each of the possible retrieval description operator classes that have been defined, including any domain-specific retrieval description operator classes. Each retrieval description operator class has an associated test function which takes a retrieval description, case feature, or feature facet as an argument, depending on that class's generic superclass, and, if possible, returns a retrieval description operator that is an instance of that class.

The following procedure tests whether a node is a goal node by matching its retrieval description to the KB, via the PARKA Structure Matcher. If a match is found, the node is a goal node.

```
Procedure goal-node (n : RDS node)

variables
   rdop : retrieval description operator of n
   old-rdesc : retrieval description of n
   new-rdesc : retrieval description
   bindings-sets : a set of variable bindings sets

begin
   new-rdesc := apply rdop to old-rdesc
   bindings-sets := match-rdesc(new-rdesc)
   if bindings-sets then
      begin
         save bindings sets with new-rdesc
         return ``true'' as result
      end
   else
      return ``false'' as result
   end if
end
```

A retrieval description operator is applied by taking an old retrieval description and modifying it by adding, deleting, or replacing one or more of its case features or feature facets, as specified by the information in the retrieval description operator. The result is a new retrieval description.

The procedure **match-rdesc** attempts to match the retrieval description associated with node *n* to the KB via the PARKA Structure Matcher. The retrieval description is converted to a probe graph (or probe graphs) which are processed by the PARKA Structure Matcher.

If the probe graph is successful, variable bindings are returned (with duplicates removed). The procedure returns NIL otherwise.

Procedure **rdesc-search** has the following parameters that can optionally be supplied for a particular kind of item (e.g., a source plan) to be retrieved and for a particular domain:

- the function for generating successor RDS nodes
- the function for generating an RDS node's score
- the function for ranking RDS nodes in the open list
- the function for evaluating a match to a retrieval description
- whether to ignore RDS nodes seen before on the open list

4.6.4 Retrieval Description Search Example

The following is a more detailed description of the retrieval description search in the example of Section 4.2.

Recall that *RDESC-1*, the initial retrieval description created by the Problem Analyzer contains a single case feature, *Feature-3.1* (Section 4.4.1). This retrieval description describes the source plan being sought: a plan for delivering a liquid package from *City1-CL1* to *City1-CL2*.

Procedure **rdesc-search** initializes the RDS open list to a new node, NODE-0 (RDESC-1). NODE-0 is chosen for expansion from the open list and RDESC-1 is converted to a probe graph, shown in Figure 4.3. The probe graph for RDESC-1 fails (no match is found), and thus NODE-0 is not a goal node.

Next, applicable retrieval description operators are sought for the case features and feature facets of *RDESC-1*. *Feature-3.1* has 6 facets: Package, Origin, Destination, Goal, Plan, and Situation. As discussed in Section 4.2, the applicable retrieval description operators RDOP-1, RDOP-2, and RDOP-3 are Variabilize-Facet(Origin), and Variabilize-Facet(Destination), Generalize-Facet (Package), respectively.[4] They are instantiated by procedure **node-successors** in RDS nodes NODE-1, NODE-2, and NODE-3 respectively. The retrieval description search tree generated is shown in Figure 4.8.

As RDOP-1 applies to a package and the other retrieval description operators apply to the origin and destination, RDOP-1 is preferred (using a domain heuristic that prefers changing the package over the origin and destination). Thus NODE-1 is ordered first on the open list and therefore chosen for expansion. RDOP-1 is applied to RDESC-1 to generate a new retrieval description, RDESC-2. RDESC-2 has a single case feature Feature-3.1a, a copy of Feature-3.1, with the Package facet generalized from type *Package* to type *Liquid-Package*.

---

[4] The modification of generalizing the goal type is not considered because achieving a goal other than "located at" (e.g., "located near") will not solve the target problem.

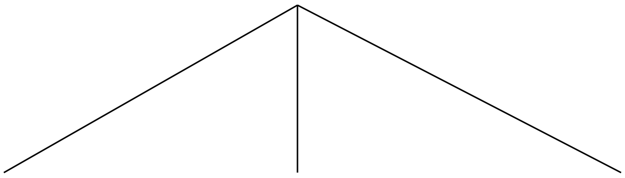

Figure 4.8: A sample retrieval description search tree.

NODE-1 is chosen for expansion from the open list and RDESC-2 is converted to a probe graph, shown in Figure 4.5. The probe graph for RDESC-2 succeeds and thus NODE-1 is a goal node and **rdsec-search** terminates with the following two sets of query variable bindings:

```
(?goal . Plan-CE-Located-At-1.3-Goal-1)
(?plan . Plan-CE-Located-At-1.3)
(?pkg-pvar . PVAR-P1)
(?sitval . SitVal-1405)
(?isit . Case-Make-Deliveries-1-InitSit)
(?pkg . Pkg-1.1)
(?dest . PVAR-L12B)

(?goal . Plan-CE-Located-At-1.2-Goal-1)
(?plan . Plan-CE-Located-At-1.2)
(?pkg-pvar . PVAR-P1)
(?sitval . SitVal-1405)
(?isit . Case-Make-Deliveries-1-InitSit)
(?pkg . Pkg-1.1)
(?dest . PVAR-L12B)
```

In the first set, query variable ?PLAN to bound to *Plan-Located-At-1.3*, the top-level plan of case *Case-Make-Deliveries-1*. In the second set, query variable ?PLAN is bound to *Plan-Located-At-1.2*, a subplan of *Plan-Located-At-1.3* (see Figure 3.9). The more specific plan *Plan-Located-At-1.2* is chosen over its parent plan *Plan-Located-At-1.3*. Thus the first source plan used will be *Plan-Located-At-1.2*. This plan is shown in Section 3.4.2 and in detail in Appendix C.2. It is passed to the Plan Mapper for further processing.

4.6.5 Discussion

The process of searching a space of retrieval descriptions was initially developed to overcome the limitation that the PARKA Structure Matcher only returns exact matches for a probe graph. The relaxation of the probe is thus the responsibility of CAPER. A potential modification of the PARKA Structure Matcher would be to relax, in parallel, the probe simultaneously along multiple dimensions (perhaps taking account of the cardinalities of the KB relations involved).

This modification might make the process faster than the current iterative serial process in which CAPER relaxes the probe and then passes it back to the PARKA Structure Matcher for processing. This modification, however, would mean giving up some control over the probe modification process and perhaps a loss of flexibility in applying various domain-specific heuristics, etc. A proposed extension to CAPER's current query modification process would make it more knowledge-intensive by treating the formation of retrieval descriptions (retrieval plans) as a planning task (see Section 9.1.3).

Some potential problems with automatic probe relaxation in parallel by the PARKA Structure Matcher could occur if the probe relaxation were to occur by automatically replacing a node in the probe graph with one of its ancestors (or descendants). The algorithm would have to handle the case where a node has multiple parents (children) and thus has multiple generalizations (specializations). This is problematic when the is-a abstraction hierarchy is not built specifically for the CBP system.

For example, consider the piece of a hypothetical general-purpose semantic network shown in Figure 4.9. Suppose we have a target problem of pounding in a nail. We then retrieve a similar case from the casebase where the same goal was achieved using a rock, *Rock-1*. We now need to find the analogue of the rock in the target problem. First we look for another rock via a probe/query that contains the constraint *isa(x,Rock)* as one of its constraint clauses. Suppose this fails because no rock is available in the current situation. Thus a non-rock substitute must be found. The rock constraint is generalized by replacing *Rock* with one or the other of its supercategories, *Pounding-Tool* or *Geographic-Feature*. We get the following two alternative constraints for the query: *ancestor(x,Pounding-Tool)* and *ancestor(x,Geographic-Feature)*. Obviously the former constraint is what we want *in this context* and will match instances of *Hammer* (e.g., *Hammer-1*). The latter constraint will lead us astray by matching instances of *Brook* (e.g., *Brook-1*), which are hardly usable for the task at hand.

An related problem comes when distances in an is-a hierarchy are used as a measure of similarity. Similarity becomes highly sensitive to how someone has set up the semantic network, particularly if he or she used redundant links (Kolodner, 1993).

4.6.6 Retrieval Plans

A retrieval plan is a data structure that holds a sequence of retrieval descriptions that have been (or will be) used to retrieve a relevant source case from memory for a target problem. The use of retrieval descriptions to specify the kind of information (cases) sought

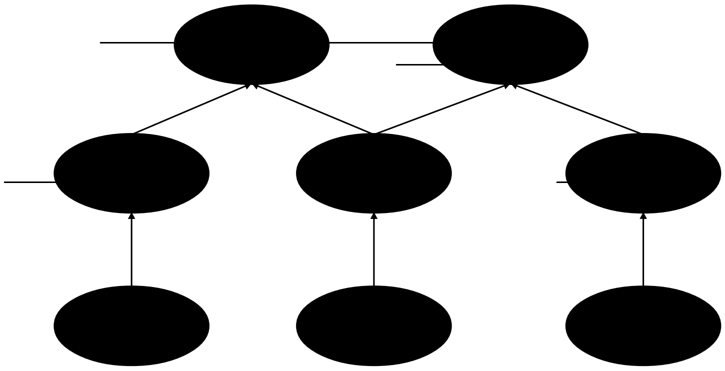

Figure 4.9: Part of a hypothetical general-purpose semantic network.

from memory was inspired by work in the psychology of human memory (Section 8.3.1). A retrieval description is an "action" in a retrieval plan. A retrieval description is "executed" as a probe graph via the PARKA Structure Matcher. Details about the contents of retrieval plans and an example are given in Appendix B.1.6.

In addition to the retrieval description search process described above, a complete domain-specific retrieval plan can be supplied by a domain engineer. The retrieval plan is followed by executing its retrieval descriptions in turn until it succeeds or all of its retrieval descriptions have been tried. This was done in an earlier version of CAPER for the Car Construction domain (Section A.2). In this domain, CAPER first attempted to find a plan in the KB for building a car with the target engine type (e.g., 4 cylinder), body style (e.g., sedan), and top type (e.g., convertible). If the probe fails, it was generalized using domain-specific knowledge and the abstraction hierarchy. For example, *4 cylinder car engine* was generalized to *car engine*. (See (Kettler *et al.*, 1994) for a complete example).

If probe was successful, another probe is issued to find a plan for building a car with a target option (e.g., air-conditioning). The results of these probes (and subsequent ones) would be combined to yield a source plan for building a car most like the target one. This probe generalization process was less flexible than the process of applying retrieval description operators to retrieval descriptions that is used by the current version of CAPER.

4.7 Retrieval Times

CAPER's approach is predicated on the availability of fast retrieval methods which scale well to large casebases. This section shows some of the empirical tests we have done on the processing of probe graphs by the PARKA Structure Matcher (Section 3.1.4). Fast retrieval time is important in CAPER because a lot of time is spent in memory. For the running example problem used throughout this chapter and the following two chapters, the breakdown for time spent in memory is shown in Table 4.1. For the small casebase used (2 cases, 1458 frames, 14,000 assertions), CAPER spent 55% of its problem solving time in memory (i.e., waiting for PARKA to process queries).

Type	**Count**	**Time**	**% Query Time**	**% Total Time**
Bottom Up	3507	28	40	22
Top Down	11	5	7	4
Recognition	15	2	2	1
Structure	15	36	51	27
All	3548	71	100	55

Table 4.1: Query Time Breakdown for a Sample Problem. ("Type" is the PARKA query type. Times are in seconds. Total problem-solving time was 125 seconds. Average number of conjuncts for recognition and structure queries was 7.5. The Common Lisp version of Parka 3 was used on a serial Macintosh Quadra 800.)

Using parallel retrieval methods can dramatically reduce the time to process queries and thus speed up overall processing. Furthermore, as shown below, these methods scale well to larger casebases.

4.7.1 Results from the UM Translog Domain

To test the efficiency and scalability of retrieval using the PARKA Structure Matcher for CAPER, several sample queries (probe graphs) were timed on the alpha version of Parka 3 for the Sun Microsystems Sparc machines and the IBM SP2, a high-performance (MIMD) parallel machine with multiple processor nodes.[5] The purpose of these experiments was to determine the absolute time for retrieval probes (queries) and to determine the scalability of parallel retrieval methods to large casebases. Several queries were run on several casebases, of varying size.

The test queries, chosen to be representative of those issued by the planner, were (in their English equivalents):

Q1: find all plans for transporting a package between locations in different cities connected by a direct route
Q2: find all plans using a particular train station
Q3: find all top-level plans using a regular-type truck
Q4: find all plans for getting an object between locations in the same city
Q5: find all plans for getting an object between locations in different cities
Q6: find all compatible packages and vehicles

---

[5] These timings were made by Kilian Stoffel.

Non pre-indexed casebases of the various sizes were used. The number of cases, plans, and frames for each is shown in Table 4.2.[6] Cases consist of plans generated by the UM NONLIN planner on randomly generated problems.

Cases	**E-plans**	**Frames**	**Assertions**
20	643	10246	129888
100	4071	58344	899132
200	8130	116728	1746060

Table 4.2: Casebases of UM TRANSLOG plans used for sample queries.

The above queries were run in serial on a Sun Sparc for the 20, 100, and 200 case casebases. These results are shown in Figure 4.10. These queries were also run for the 200 case casebase on the IBM SP2 with varying numbers of nodes (processors) on the 200 case casebase. These results are shown in Figure 4.11.

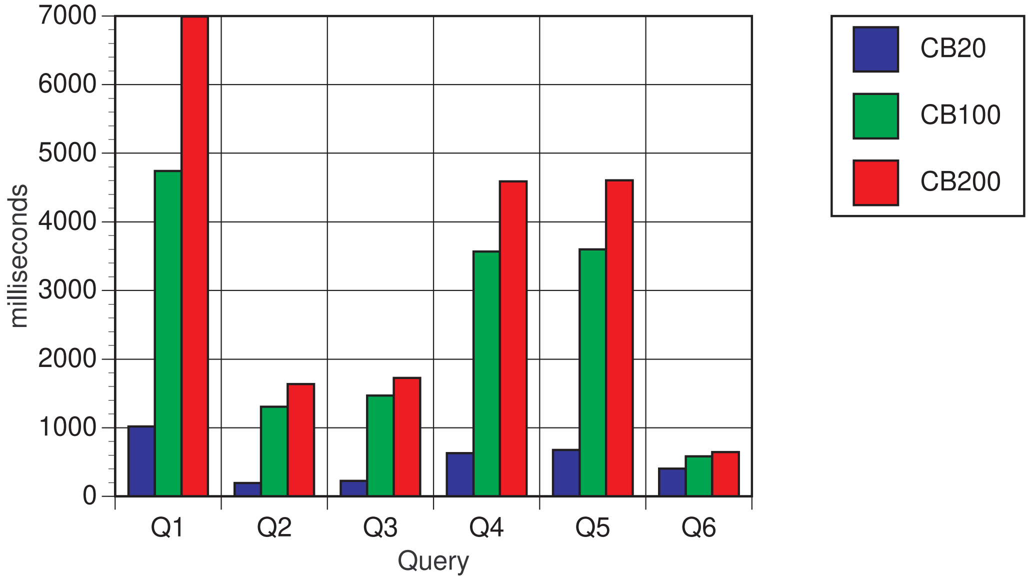

Figure 4.10: Serial retrieval times for sample UM TRANSLOG queries on a Sun Sparc on several casebases.

These results show absolute parallel times under half a second on the largest casebase. Clearly these times are fast and the total cost of frequent memory accessing will be small. The cost of frequent memory accessing using the serial implementation, in contrast, would

---

[6] Note that the number of plans is important because each plan can be retrieved independently and is the functional equivalent of a case in a CBP system that stores cases monolithically. Frame and assertion counts are shown for the Parka 3 Common Lisp version.

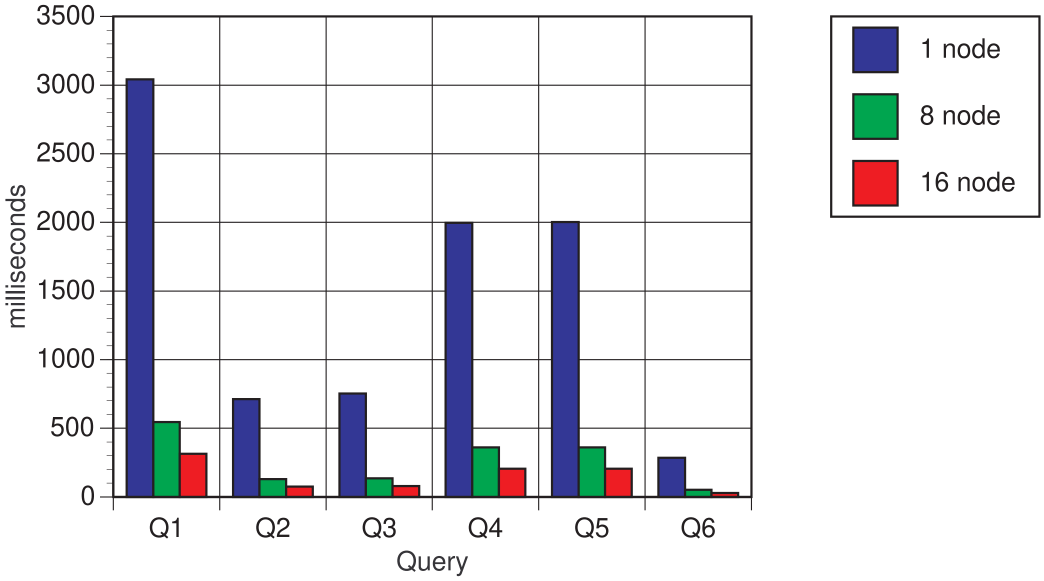

Figure 4.11: Parallel retrieval times for sample UM TRANSLOG queries on the IBM SP2 in 1, 8, and 16 node (processor) configurations on the 200 case casebase.

be fairly expensive. These results also show that parallel times scale much better to larger casebases than do the serial ones, assuming a sufficient number of parallel processors are available.

4.7.2 Results from the CMU Translog Domain

The data shown in this section are from a version of the CMU transport logistics domain (Veloso, 1992), which was used to test an earlier version of CAPER. The probe graphs were processed by a previous version of PARKA (v2.0) running on the massively parallel Connection Machine 2 (CM-2). To test the efficiency and scalability of retrieval using the PARKA Structure Matcher for CAPER, several sample queries (probe graphs) were timed. The purpose of these experiments was to determine the absolute time for retrieval probes (queries) and to determine the scalability of parallel retrieval methods to large casebases. For an analysis of the complexity of the PARKA Structure Matcher, see (Andersen *et al.*, 1994).

The test queries, chosen to be representative of those issued by the planner, were (in their English equivalents):

Q1: find all top-level plans involving 2 or more packages
Q2: find all plans using a particular airport
Q3: find all top-level plans using 2 or more trucks
Q4: find all plans for getting an object between locations in the same city

Q5: find all plans for getting an object between locations in different cities
Q6: find all top-level plans where a truck and an object are in the
same location in the initial situation

For example, one query (Q2) was to find all plans using a particular airport. This query might be useful in avoiding plans that used an airport that is closed in the current situation. The sample transport logistics queries are considerably more structurally and computationally complex than those used for the car construction domain. The added computational complexity is due to combinatorial effects in matching objects in the plans to probe variables.[7] These queries were issued as probe graphs for the PARKA Structure Matcher.

Non pre-indexed casebases of the various sizes were used. The number of cases, plans, and frames for each is shown in Table 4.3.[8] Cases consist of plans generated by the UM NONLIN planner on randomly generated problems.

Cases	**E-plans**	**Frames**
20	296	3357
40	523	4660
60	740	5899
80	982	7280
100	1213	8616

Table 4.3: Casebases of CMU transport logistics plans used for sample queries.

Queries were run using PARKA running on a CM-2 with 32,000 virtual processors (16,000 physical processors) and also using a highly optimized serial implementation of PARKA.[9] Note that a *direct* comparison of the time for a particular query/casebase run in serial to the same query/casebase in parallel is *not* meaningful. The results are shown in Figures 4.12 and 4.13.

These results show absolute parallel times ranging from 0.5 to 1.6 seconds, on the larger casebase. Clearly these times are fast and the cost of frequent memory accesses will be small. The cost of frequent memory accesses using the serial implementation, in contrast, would be fairly expensive.

These results also show that parallel times scale much better to larger casebases than do the serial ones. The performance of the serial procedures was clearly worse than linear in

---

[7] For example, matching a probe for the query “find all plans in which two packages were delivered” against a stored plan involving five packages produces 20 possible variable bindings of the two probe variables for packages.

[8] Note that the number of plans is important because each plan can be retrieved independently and is the functional equivalent of a case in a CBP system that stores cases monolithically.

[9] Written in Macintosh Common Lisp and running on a Macintosh IIcx.

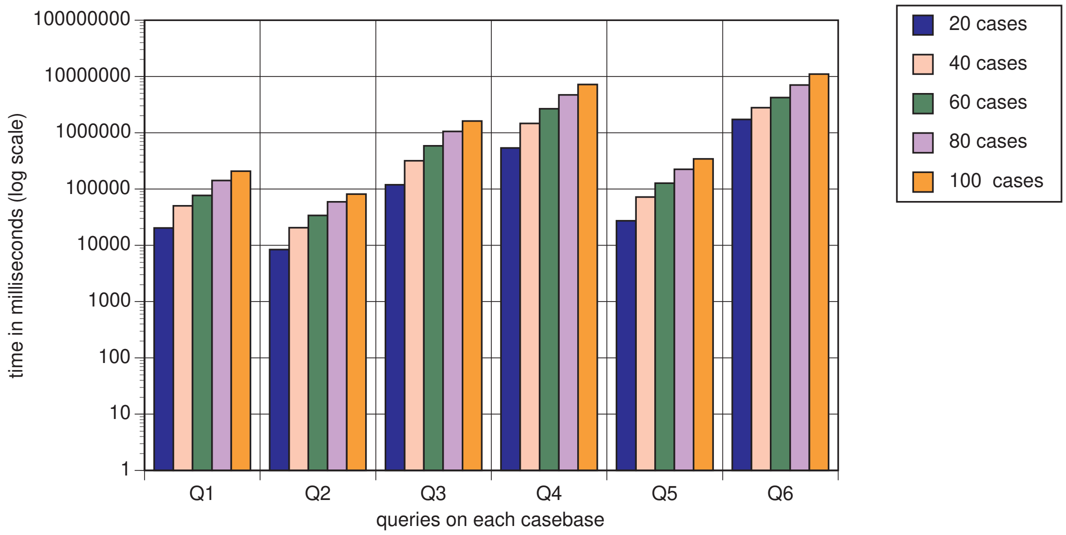

Figure 4.12: Serial retrieval times for sample CMU transport logistics domain queries (shown on a logarithmic scale)

the size of the casebase. The parallel procedures, in contrast, exhibited times growing better than logarithmically in the size of the casebase. Thus, using parallel methods, casebases in the thousands of cases can be accessed efficiently, assuming a sufficient number of processors are available.

Results similar to those above were also found for the automobile assembly domain (Kettler *et al.*, 1993). While a serial implementation using an *pre-indexed* memory might perform better than our serial implementation O(log n) in the number of cases if a balanced discrimination tree were used, such a system would suffer from the problems described in Section 2.3.3.

4.8 Chapter Summary

This chapter discussed the following:

- The Problem Analyzer component creates a higher-level representation of the target planning problem, i.e., a problem description consisting of case features (with feature facets). This is in addition to the lower-level, graph-structured representation of cases as nodes in the semantic network (the persistent KB).

- The Plan Retriever component generates an initial retrieval description from this problem description, processes the retrieval description via the Retriever, and ranks and returns the source plans retrieved.

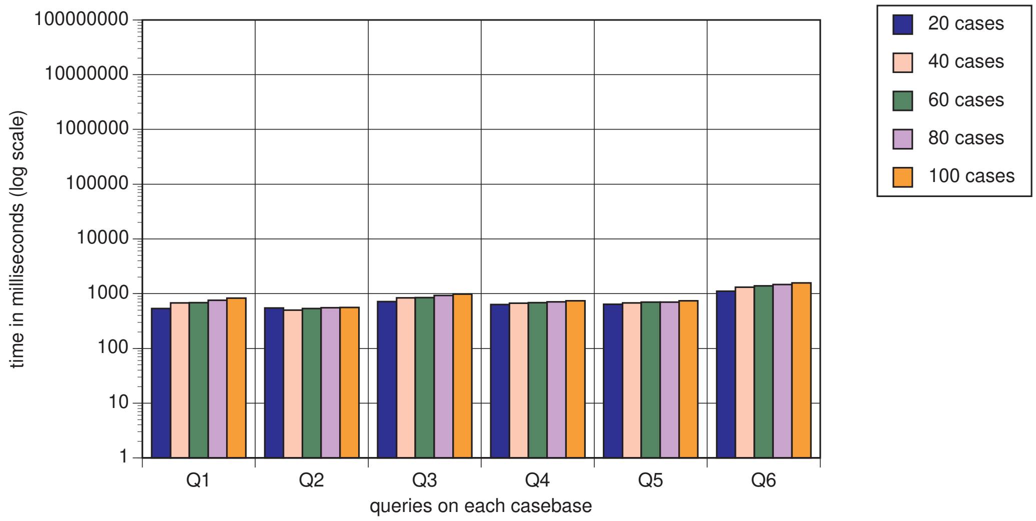

Figure 4.13: Parallel retrieval times for sample CMU transport logistics domain queries (shown on a logarithmic scale)

- The Retriever component takes a retrieval description and converts it to PARKA Structure Matcher probe graph. If the probe graph is unsuccessful, the retrieval description is modified using retrieval description operators. Thus the Retriever searches a space of retrieval descriptions until a successful one is found.
- The Retriever typically accesses memory frequently, at least once for each retrieval description.
- Retrieval is flexible because the retrieval description can contain any feature of the target problem. In other words, because a probe graph may contain any nodes or links, a plan (or part of a plan) can be retrieved through any of its constituent nodes.

Chapter 5

Elaboration in CAPER

5.1 Overview

The process of elaboration in analogical reasoning consists of mapping entities and relations between the source analogue and the target analogue in order to transfer knowledge between them. In CAPER, elaboration consists of establishing a complete global mapping between the source case and target problems and instantiating a target plan from a source plan using this mapping. A local mapping is a correspondence between an entity (or relation) in one analogue to an entity in another. A global mapping is a set of all local mappings. Usually it is desirable for a global mapping to be structurally isomorphic: one-to-one and consistent. Elaboration occurs after a plan has been retrieved using the process described in the preceding chapter. The target plan that is instantiated will typically have to be modified by the process described in the next chapter.

The process of elaboration in CAPER involves a kind of plan adaptation known as *reinstantiation* (Kolodner, 1993). The task of reinstantiation is to find alternate role bindings. Roles are bound to one set of values in the source case and must be rebound to a (potentially different) set of values in the target case. This task requires "knowing (1) the correspondences between roles in the old case and the new case and (2) an abstracted (variabilized) framework for the old solution." (ibid, 399). The steps involved in reinstantiation are:

1. **Abstract** the framework of the old problem and solution.

2. **Compute correspondences** between the roles in the two problem statements.

3. **Instantiate** the framework of the old problem in the new case and solution based on computed correspondences.

CAPER has an **abstracted framework** for an old plan via its use of roles and plan variables. Plans created by CAPER (or by the Casebase Seeder using UM NONLIN) have their roles filled by plan variables, which are in turn bound to constants in their respective cases. For example, assume plan variable v is bound to constant c_1 in plan p_1 and fills roles $r_1, r_2, \ldots, r_n$. When v is reinstantiated for plan p_2 in a *different* case, the value c_2 it is bound to must fill roles $r_1, r_2, \ldots, r_n$.

The **correspondences** between the roles in the problem statements are fairly easy to determine in CAPER. Some of these are initially determined during plan retrieval. In the example of Section 4.2, we know that *Pkg-1.1* in *Case-Make-Deliveries-1* corresponds to *Pkg-3.1* in *Case-Make-Deliveries-3* because they both fill the same role (package) in the corresponding plan goals. The remaining correspondences are computed by the Plan Mapper using several techniques including matching role fillers to each other using constraints from the source case.

The abstracted framework is **instantiated** using these correspondences by the Plan Instantiator.

In brief, elaboration in CAPER involves the following modules:

- The **Plan Mapper** (Section 5.3) takes as input a source case, retrieved by the Plan Retriever (Section 4.5), and a set of local mappings established during plan retrieval (or by previous invocations of the Plan Mapper). It returns a complete global mapping between the source case and target problem. It does this using a forward-checking constraint satisfaction algorithm to bind source plan variables from the source case to entities in the target problem. The constraints come from plan validations, which capture the source plan dependencies involving plan variables and the entities they are bound to.

- The **Plan Instantiator** (Section 5.4) takes a source plan and the global mapping created by the Plan Mapper and returns a target plan. This target plan is a reinstantiation of the source plan with source values having been replaced by target values.

Before the discussion of the data and modules involved in plan elaboration, a simple example will be described in the following section.

5.2 An Example

This section continues the UM TRANSLOG domain (Section A.1) example of Section 4.2 in which the target problem of Case-3 was to deliver a non-hazardous, non-valuable liquid package, *Pkg-3.1*, from *City1-CL1* to *City1-CL2*. A source plan *Plan-Located-At-1.2* from case *Case-Make-Deliveries-1* (Case-1) was retrieved by the Plan Retriever. This source plan has the following thirteen actions:

```
1. Do-Nothing-1.1 (type: Do-Nothing)
2. Do-Nothing-1.2 (type: Do-Nothing)
3. P-Collect-Fees-1.1 (type: P-Collect-Fees)
      package: PVAR-P1 (Pkg-1.1)
4. Do-Nothing-1.3 (type: Do-Nothing)
5. P-Open-Door-1.1 (type: P-Open-Door)
      vehicle: PVAR-T1 (Truck-1)
6. P-Load-Package-1.1 (type: P-Load-Package)
      package: PVAR-P1 (Pkg-1.1)
      vehicle: PVAR-T1 (Truck-1)
```

```
 7. P-Close-Door-1.1 (type: P-Close-Door)
       vehicle: PVAR-T1 (Truck-1)
 8. P-Move-Vehicle-1.1
       vehicle:     PVAR-T1   (Truck-1)
       origin:      PVAR-L11C (City1-Cl1)
       destination: PVAR-L12B (City1-Cl2)
       route:       P-Local-Road-Route
 9. P-Open-Door-1.2 (type: P-Open-Door)
       vehicle:     PVAR-T1   (Truck-1)
10. P-Unload-Package-1.1 (type: P-Unload-Package)
       package: PVAR-P1 (Pkg-1.1)
       vehicle: PVAR-T1 (Truck-1)
11. P-Close-Door-1.2  (type: P-Close-Door)
       vehicle:     PVAR-T1   (Truck-1)
12. Do-Nothing-1.4 (type: Do-Nothing)
13. P-Deliver-1.1 (type: P-Deliver)
       package: PVAR-P1 (Pkg-1.1)
```

Shown in parentheses are the bindings for plan variables. Details on this plan are given in Appendix C.2.

The elaboration routines in CAPER must reinstantiate this source plan as a target plan for Case-3. First, a mapping must be found between role fillers in the source plan and entities in the target case. Role fillers include plan variables (Section 3.4.4) and constants. A plan variable with value c_s in the source case must be reinstantiated with value c_t in the target case, where c_s and c_t are analogues. (It is also possible that $c_s = c_t$.)

The plan variables to be reinstantiated from plan *Plan-Located-At-1.2* are the following:

Plan Variable	**Source Value**	**Roles**
PVAR-P1	Pkg-1.1	package to deliver
PVAR-RP	Regular-Package	type of package bound to PVAR-P1
PVAR-L11B	City1-CL1	package origin
PVAR-L12B	City1-CL2	package destination
PVAR-T1	Truck-1	truck used in delivery
PVAR-RT	Regular-Truck	type of vehicle bound to PVAR-T1
PVAR-L11A	City1-CL1	location of vehicle bound to PVAR-T1
PVAR-L11C	City1-CL1	location of vehicle bound to PVAR-T1
PVAR-C1A	City1	city of PVAR-11C and PVAR-12B
PVAR-C1B	City1	city of PVAR-11B and PVAR-12B

In the source plan *Plan-Located-At-1.2*, plan variable *PVAR-T1* designates the vehicle used to transport the package. In the source case, this variable is bound to *Truck-1*, a regular truck. The mapping procedure needs to find an object (constant) c_t in the target problem that can be bound to PVAR-T1. That is, an analogue c_t for *Truck-1* needs to be found in the target situation. The value c_t must play the roles and satisfy the constraints in the target problem analogous to those for *Truck-1* satisfies in the source problem.

When a source plan is retrieved, it must match some case feature (Section 4.4.1) of the target problem. Some case feature in the retrieval description (Section 4.6.1) matches some source case feature. From this correspondence between case features some initial mappings are derived. If facet f has value c_t for the target feature and value c_s for the analogous source feature, then c_t maps to c_s (and vice versa).

For example, the Case-3 feature *Feature-3.1* was matched to a Case-1 feature *Feature-3.1s* by the Plan Retriever. The plan variables of *Plan-Located-At-1.2* with corresponding target values and relevant feature facets are:

Plan Variable	**Target Value**	**Facet**
PVAR-P1	Pkg-3.1	Package
PVAR-RP	Liquid-Package	Package Type
PVAR-L11B	City1-CL1	Origin
PVAR-L12B	City1-CL2	Destination

The task is now to find target values for the remaining plan variables. This is done using **plan variable constraints**. These constraints arise from the target plan validations (Section 3.4.6). The validations capture dependencies of the plan upon objects in the world.

Figure 5.1 shows the seven constraints on plan variable *PVAR-T1*, bound to *Truck-1* in the source case. These constraints are derived from ten validations that hold in the Case-1's initial situation. For example, the *at* link from *PVAR-T1* to *PVAR-L11A* corresponds to the constraint that the vehicle bound to *PVAR-T1* must be at the location bound to *PVAR-L11A*. This binary plan variable constraint comes from plan validation 20 in *Case-1*, which has the condition *hasType(PVAR-T1,PVAR-L11A)*. (Constraints are numbered after the lowest numbered source validation they correspond to.) A target value for a *PVAR-T1* should satisfy as many of these constraints as possible in order to reduce the amount of adaptation that must be done. The complete plan variable constraints created from the plan validations of *Plan-Located-At-1.2* are shown in Appendix C.2.3.

Given a set of plan variables to rebind in the target case and their plan variable constraints, the Plan Mapper invokes a constraint satisfaction procedure to find a set of variable bindings that satisfies the constraints. Constraint checking involves frequent accessing of the KB.

Each plan variable is assigned an initial domain, a set of values that the variable can take as its value. Initial domains are computed using domain-specific knowledge.

For example, the value of *PVAR-T1* has to be a vehicle. One cannot replace its source value *Truck-1* by an airport, for example, because the target plan cannot have actions such as "Move Vehicle Airport-1 from City1-CL1 to City1-CL2". Thus a plan variable bound to a vehicle in the source case must be bound to a vehicle in the target case.

Additional plan variable constraints will serve to specify the type(s) of vehicles that are permissible. Plan variable constraints are of two types: unary and binary. Unary plan variable constraints involve a single *unbound* plan variable. Binary plan variable constraints involve two *unbound* plan variables. In Figure 5.1, for example, plan variable constraint 10, *available(PVAR-T1,True)*, is a unary constraint. Constraint 1, *hasType(PVAR-T1,PVAR-RT)*, is a binary constraint.

Unary constraints are applied to narrow the domain of each plan variable. The details of this process are described in Section 5.3.2. For a given plan variable, all of its unary

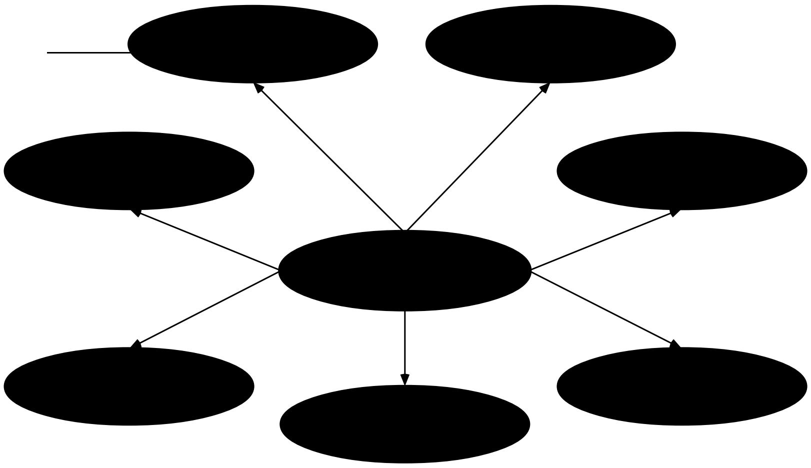

Figure 5.1: Constraints on plan variable PVAR-T1. (Variables begin with "PVAR" and are bound in the *source* case to the constants shown beneath them inside the ovals (nodes). The numbers of the Case-1 plan validations associated with each constraint (arc) are given in parentheses after the link name.)

constraints are checked and domain values that satisfy a particular constraint have their scores increased. Unary constraints are checked by issuing PARKA queries to find out which domain values satisfy the constraint.

For example, constraint 10, *available(PVAR-T1,True)*, is checked via the query *available(?x,True)*, where ?x is a query variable.[1] The bindings returned for ?x are *Truck-1* and *Truck-3*, i.e., those trucks available in the initial situation of Case-3. Because only these trucks can be used in the target plan, all other vehicles are pruned from the domain of *PVAR-t1* and the scores for *Truck-1* and *Truck-3* are incremented by 1. (As *Truck-1* is a regular-type truck, it will later be found unsuitable for carrying the liquid-type package *Pkg-3.1*. This is described below. *Truck-3* is a tanker truck and thus suitable for carrying the target package.)

If a domain of a plan variable is reduced to a single value, that value is chosen as the target value for the variable. This choice is propagated through the binary constraints that involve that variable.

After all unary constraints have been checked for each unbound plan variable, the Plan Mapper attempts to satisfy the remaining binary constraints for each variable. Variables are checked in decreasing order of the number of binary constraints they have, i.e., a plan variable with the most binary constraints will be checked first, etc. This is a typical constraint-satisfaction heuristic.

---

[1] The actual query is more complicated than this as *available* is a situated slot Section 3.3: i.e., its filler depends on the situation which here is *Case-Make-Deliv-1-InitSit*.

To satisfy the binary plan variable constraints, the Plan Mapper uses a forward-checking constraint-satisfaction algorithm (with graph search) described in detail in Section 5.3.3. Binary constraints are checked using PARKA Structure Matcher queries. The values in the domain that satisfy a binary constraint are either pruned or have their scores incremented (depending on particular constraint involved).

Figure 5.2 shows some of the the constraints involving the unbound variables PVAR-T1 and PVAR-RT to be satisfied. To satisfy these constraints, a truck must be chosen for the value of *PVAR-T1* that has a type (bound to *PVAR-RT*) that is compatible with the package type *Liquid-Package*.

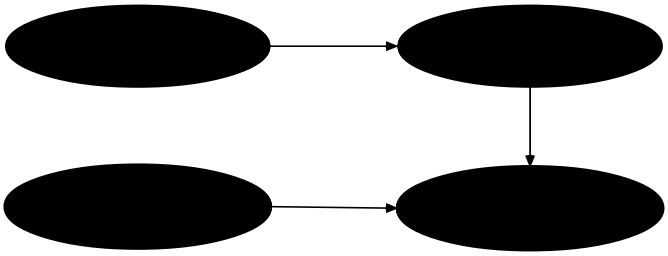

Figure 5.2: Some constraints to be satisfied for two unbound plan variables. (Plan variables PVAR-P1 and PVAR-RP have already been bound to the target values *Pkg-3.1* and *Liquid-Package*, respectively.)

There are two values in the domain of *PVAR-T1* at this point: *Truck-3* and *Truck-1* with scores of 3 and 2, respectively. The forward-checking constraint satisfaction routine tries each value in the domain, in order of decreasing score, to see if by picking that value for the variable, at least one valid values remains in the domain for each of the other plan variables yet to be bound.

For example, if *Truck-3* is chosen for the value of *PVAR-T1*, its type *Liquid-Package* can be chosen as the value of *PVAR-RT*. If *Truck-1* is chosen for the value of *PVAR-T1*, however, its type *Regular-Truck* cannot be chosen as the value of *PVAR-RT*, because package type *Liquid-Package* is *not* compatible with vehicle type *Regular-Truck*. Thus *Truck-3* and *Tanker-Truck* will be chosen as the target values for *PVAR-T1* and *PVAR-RT*, respectively.

The other plan variables are bound in a similar manner. The end result is the following global mapping:

Source Var.	**Source Value**	**Target Value**	**Target Var.**
PVAR-P1	Pkg-1.1	Pkg-3.1	PVAR3-PKG
PVAR-RP	Regular-Package	Liquid-Package	PVAR3-PKGT
PVAR-L11B	City1-CL1	City1-CL1	PVAR3-ORGN
PVAR-L12B	City1-CL2	City1-CL2	PVAR3-DEST
PVAR-T1	Truck-1	Truck-3	PVAR3-VEH
PVAR-RT	Regular-Truck	Tanker-Truck	PVAR3-VEHT
PVAR-L11A	City1-CL1	City1-CL1	PVAR3-LOCA
PVAR-L11C	City1-CL1	City1-CL1	PVAR3-LOCC
PVAR-C1A	City1	City1	PVAR3-CTYA
PVAR-C1B	City1	City1	PVAR3-CTYB

The target variables shown are newly instantiated plan variables that will fill roles in target plan analogous to the ones filled by their corresponding variables in the source plan.

The following target plan, *TPLAN-3*, is instantiated for *Case-Make-Deliveries-3* by the Plan Instantiator (Section 5.4):

```
 1. Do-Nothing-3.1
 2. Do-Nothing-3.2
 3. P-Collect-Fees-3.1
      package: PVAR3-PKG (Pkg-3.1)
 4. Do-Nothing-3.3
 5. P-Open-Door-3.1
      vehicle: PVAR3-VEH (Truck-3)
 6. P-Load-Package-3.1
      package: PVAR3-PKG (Pkg-3.1)
      vehicle: PVAR3-VEH (Truck-3)
 7. P-Close-Door-3.1
      vehicle: PVAR3-VEH (Truck-3)
 8. P-Move-Vehicle-3.1
      vehicle:     PVAR3-VEH (Truck-3)
      origin:      PVAR-LOCA (City1-Cl1)
      destination: PVAR-DEST (City1-Cl2)
      route:       P-Local-Road-Route
 9. P-Open-Door-3.2
      vehicle: PVAR3-VEH (Truck-3)
10. P-Unload-Package-3.1:
      package: PVAR3-PKG (Pkg-3.1)
      vehicle: PVAR3-VEH (Truck-3)
11. P-Close-Door-3.2
      vehicle: PVAR3-VEH (Truck-3)
12. Do-Nothing-3.4
13. P-Deliver-3.1
      package: PVAR3-PKG (Pkg-3.1)
```

Notice that the target action names have been modified and that source plan variables have been replaced by new target plan variables whose values are those established above.

This plan is passed to the Plan Modifier, described in the next chapter.

5.3 The Plan Mapper

This section presents details about the Plan Mapper, whose operation was illustrated in the example of the previous section. The Plan Mapper module attempts to find a global mapping between the source and target cases. Input to the Plan Mapper is:

- the target problem description
- the source plan retrieved by the Plan Retriever
- the mapping between between source and target case features found during plan retrieval (and stored in source plan descriptor (Section 4.5.2))
- domain-specific parameters (to be described below)

Output from the Plan Mapper is a global mapping and a newly instantiated target plan.

The global mapping contains a set of local mappings. In general, given plan variable v bound to constant value c_s in the source case, the mapper must find a corresponding constant value c_t in the target case. It is possible that $c_s = c_t$. Constants may be objects, object types, predicates, etc. that are defined in the KB. A local mapping is a 3-tuple $\langle v, c_s, c_t \rangle$, where c_s and c_t are analogous. A plan variable descriptor is also returned for each plan variable that was reinstantiated.

5.3.1 Initial Procedures

Procedure **map-plan** is the main routine for the Plan Mapper. It is invoked by CAPER's Main module after the chosen source plan has been retrieved by the Plan Retriever. This source plan is either a plan to apply to the top-level ("root") target plan or a patch/substitute plan to repair a faulty target plan.

In the latter event, the mapping generated for the newly retrieved plan must be made consistent with the mapping for previously retrieved and mapped plans. For example, suppose variable v_1, representing the vehicle that delivers a package in source plan s_1, is reinstantiated with value *Truck-66* in the target plan t_1. Next suppose that we need to fetch an alternate subplan for loading *Truck-66*. We retrieve an appropriate source plan s_2 for loading the truck and instantiate it as target subplan t_2 (a subplan of t_1). The variable v_2 in s_1 that represents the truck being loaded *also* needs to be reinstantiated with value *Truck-66*. Otherwise, we will be loading a different truck than is used to deliver the package.

Procedure **map-plan** does the following:

1. Initializes the **domains of values** for each plan variable to be bound.

2. Refines the domains for each unbound plan variables by applying **unary plan constraints**.

3. Satisfies the **binary plan variable constraints** and returns the target values of the plan variables.

Each plan variable is assigned a initial domain of values, determined using a method specific to the planning domain.[2]

For example, *PVAR-T1* gets the domain of values which is all instances of the category *Vehicle*. This is done using the knowledge that a source case vehicle must be replaced by a target case vehicle, rather than by a location, etc.

5.3.2 Unary Plan Variable Constraint Satisfaction

Unary plan variable constraints are used to narrow the domain of values for each plan variable. A unary constraint has a single *unbound* plan variable. There are two kinds of unary plan variable constraints: hard and soft. Hard unary constraints are ones that must hold. Soft unary constraints are ones that preferably hold. Constraint types are specified as either hard or soft by a domain engineer.

For example, *PVAR-T1* has several unary constraints, as shown in Figure 5.1. These include hard constraint 10 (from plan validation 10 in *Case-1*) with the form *available(PVAR-T1,True)*, i.e., the vehicle bound to *PVAR-T1* must be available in the target problem initial situation. This constraint is hard because a vehicle must be available in order to use it in the plan. *PVAR-T1* has several soft unary constraints including number 22 (from plan validations numbers 22 and 28 in *Case-1*). This constraint has the form: *hasType(PVAR-T1,Regular-Truck)*, i.e., the vehicle bound to *PVAR-T1* should have type *Regular-Truck*. This constraint is soft because we do not want to force the type of vehicle to *Regular-Truck*, as that may be incompatible with the type of the target package.

Unary Plan Variable Constraint Checking

Unary plan variable constraints are checked by the procedure **check-vcons-unary** which does the following for each unbound plan variable p with initial domain d (a set of values):

1. Check each hard unary plan variable constraint k for v and prunes values from d that do not satisfy k.

2. Set the score for each value remaining in d to h, the number of hard unary constraints checked.

---

[2] Note that "domain" has multiple meanings here. A (plan) variable "domain" is a set of values that the variable may take. A planning "domain" is the area of planning such as transport logistics, cooking, etc. "Domain-specific" means specific to a particular planning domain.

3. Check each soft unary plan variable constraint k for v. For each constraint k, if value $c \in d$ and c satisfies k, the score for c is incremented by 1.

To find the domain values that satisfy a constraint, a probe graph corresponding to the constraint is issued. Given a unary constraint k with (validation) condition $p(v, e)$, where p is a predicate and e is a constant or bound plan variable, a PARKA query of the form $p(?x, e)$ is issued, where $?x$ is a query variable. The values satisfying the constraint are the bindings for $?x$ returned by PARKA. If b is the set of bindings, then the set of domain values satisfying k is $d \cap b$, where d is the current domain of the variable.

If a domain is reduced to a single value, that value is chosen for the reinstantiation of the variable. Unary constraints of the type $p(v, c)$, where v is a plan variable and c is a *category*, are ignored during Plan Mapper.

For example, on invocation of this procedure for *PVAR-T1*, the domain of values from **get-planvar-domains** is {Truck-1, Truck-2, Truck-3, Truck-4}. After applying hard constraint 10, domain is reduced to those vehicles that are available in *Case-3*'s initial situation: {⟨Truck-1,1⟩, ⟨Truck-3,1⟩}, shown as ⟨*c*,*s*⟩, where c is a value and s is its score (i.e., the number of constraints satisfied by the value).

Procedure **check-vcons-unary** next tests the soft unary constraints for the plan variable. Here values are not pruned from the domain but are instead scored by how many constraints they satisfy.

For example, constraint 18, *at(PVAR-T1,PVAR-LL1B)*, for *PVAR-T1* is applied (*PVAR-L11B* is already bound to *City1-CL1*). Those vehicles are located at *City1-CL1* in the initial situation of *Case-3* have their scores incremented by 1, resulting in the following variable domain: {⟨Truck-1,2⟩, ⟨Truck-3,2⟩}.

After the unary constraints have been applied by procedure **check-vcons-unary**, the plan variables to be bound are sorted in decreasing order by number of binary constraints for each that remain to be satisfied.

In the current example, the plan variables in order with their number of binary constraints are: *PVAR-T1* (6), *PVAR-L11C* (2), *PVAR-L11A* (2), and *PVAR-RT* (1).

5.3.3 Binary Plan Variable Constraint Satisfaction

The procedure **fcs**, invoked by **map-plan**, is a forward-checking constraint satisfaction routine that does graph search. It takes a set of unbound plan variables, whose target domains have been initialized by the procedures described above. Solving the constraints associated with these variables consists of finding a domain set — a set of domains (one or more values), one for each unbound plan variable. A node in the search space is a domain set. An arc connecting two nodes represents a pruning of some domain in the domain set. The procedure terminates on finding a domain set that satisfies the constraints. An example of a search graph is shown in Figure 5.3 and described in detail after the procedure descriptions.

A forward-checking constraint satisfaction algorithm can be conceptualized as a process consisting of tree search plus some degree of arc consistency checking (Nadel, 1988). Arc consistency deals with dependencies between variables in the constraints to be satisfied.

More formally, an arc $\langle v_i, v_j \rangle$ is consistent if for every value x in the current domain of variable v_i, there is some value y in the domain of variable v_j such that $v_i = x$ and $v_j = y$ are permitted by the binary constraint(s) between v_i and v_j (Kumar, 1992).

When trying a value x for variable v_i, a forward-checking constraint satisfaction algorithm checks the domains of variables $v_{i+1}, v_{i+2}, \ldots v_n$ (n is the number of variables) to make sure each domain of v_k, $k = i + 1$ to n, contains one value y_k such that $v_i = x$ and $v_k = y_k$ are permitted. In constraint satisfaction algorithms, there is typically a trade-off between the amount of tree search and the amount of checking (using some degree of lookahead) to ensure arc consistency. Forward checking tends to expand more nodes on certain kinds of problems (e.g., n-queens) but with fewer constraint checks than algorithms that do more arc consistency checking (Nadel, 1988). Thus the cost of the overall process is lower. The CAPER procedures **check-forward** and **revise**, described below, are based on algorithms presented in (Nadel, 1988).

The domains of plan variables to be bound are stored in set d. Thus d_i holds the set of domain values for plan variable v_i. An element in d_i has the form $\langle c_j, s_j \rangle$, where c_j is a value (e.g., *Truck-1*) and s_j is its score (e.g., 2). The variable k is the number of the current plan variable.

The graph search procedure searches a space of states, each of which is a different set of variable domains. A node in the search space is called an **FCS node**, to distinguish it from nodes in the semantic network. An **FCS node** n holds information about state s in the search space including

- **n.d**: the value of d, the set of domains, in state s
- **n.k**: the value of k, the current plan variable, in state s
- **n.zk**: the value of zk, the current value of v_k, in state s

The following is the graph search algorithm which finds a valid set of variable domains.

```
procedure fcs (num-pvars)

begin
    if first invocation for an splan,
       d := domain set for unmapped pvars
       open-list := make-new-fcsn with k=0, d=d
    else restore d, open-list
       node := pop(open-list)

    if node.k = num-pvars then
       return node.d
    else
       k := k + 1;
       d, empty-domain := check-forward(k,d)
       if empty-domain
```

```
          then return failure
       else fcs-expand-node(node)
          fcs(num-vars);
end
```

Procedure **fcs-expand-node**, invoked by procedure **fcs**, expands a FCS node n with $n.k = k$ When a node is expanded, its successors are generated and added to the end of the open list. The successors of n are nodes $m_1, m_2, \ldots, m_p$, where p is the number of values in the domain of v_{k+}, i.e., $n.d_{k+1} = c_1, c_2, \ldots, c_p$. The values $c_1 \ldots c_p$ are ordered by decreasing score. Node m_i has $m.k = k + 1$, $m.zk = c_i$, and $m.d_{k+1} = \{c_i\}$.

```
procedure fcs-expand-node (node);

begin
   /* create one node for each domain value
   d' := copy of d
   domain := node.d[k+1]
   domain := sort domain by decreasing value scores
   for zk in domain /* zk is value w/ score */
      add new node n to end of open-list
      n.d = d'
      n.d[k+1] := {zk}
      n.k = k+1;
      n.zk = zk;
end
```

Procedure **check-forward**, invoked by procedure **fcs**, checks forward arc consistency of v_{k-1} with $v_k, v_{k+1}, \ldots, v_n$. The procedure potentially updates d, the set of domains.

```
procedure check-forward (k d)

begin
   if k > 1
      do-for-while (f k num-pvars :test not empty-domain)
         (revise f (- k 1) d)
      return d empty-domain
end
```

Procedure **revise**, invoked by procedure **check-forward**, revises d, the set of domains, to obtain forward arc consistency from $v_i \rightarrow v_j$.

```
procedure revise(i j d)

begin
   old-domaini := d[i];
```

```
   domaini := check-vcons-binary(pvari,pvarj,old-domain1,old-domain2)
   if dom1 <> old-domain1
      change := true;
      d[i] := domaini
   if dom1 is empty, then empty-domain := true
   return d empty-domain change
end
```

Procedure **check-vcons-binary**, invoked by procedure **revise**, is similar to procedure **check-vcons-unary**, described above. Binary constraints involving two variables, v_i and v_j, are used rather than unary constraints. A binary constraint is used to return all the tuples of values $\langle c_i,c_j \rangle$ that satisfy the constraint. The domain d_i is revised as follows:

$$d_i \leftarrow d_i \cap \bigcup_k c_{i_k}$$

Like **check-vcons-unary**, **check-vcons-binary** scores the values in d_i.

An Example of Binary Constraint Satisfaction

This section presents an example of the operation of procedure **fcs**. The example is the one of Section 5.2. This section presents the details as to how the binary constraints, mentioned in the example, are actually satisfied.

When **fcs** is invoked, $n = 4$ plan variables to bind and d, the set of domains, contains the following values (computed by procedure described in Section 5.3.2).

Plan Variable	**Domain**
v_1, PVAR-T1	$d_1 = \{\langle \text{Truck-3,3} \rangle, \langle \text{Truck-1,2} \rangle\}$
v_2, PVAR-RT	$d_2 = \{\text{all truck/traincar types}\}$
v_3, PVAR-L11C	$d_3 = \{\text{all city locations}\}$
v_4, PVAR-L11A	$d_4 = \{\text{all city locations}\}$

Here we will show only the binding of v_1 and v_2. Figure 5.2 shows some of the constraints (and the numbers of the *Case-1* validations they correspond to) involving the unbound variables PVAR-T1 and PVAR-RT to be satisfied. Plan variables PVAR-P1 and PVAR-RP are already bound to *Pkg-3.1* and *Liquid-Package*, respectively. Using procedure **fcs**, we need to find a value for the unbound variables such that we have a vehicle (*PVAR-T1*) to carry the package whose type (*PVAR-RT*) is compatible with the package type (*PVAR-RP*).

Figure 5.3 shows the top of the search tree produced by procedure **FCS**. The values of k, zk, and d for each node at the time the node is generated are shown.

Initially Node 0 is created and put on the open list. It is chosen for expansion by procedure **fcs-expand-node**, which then generates its successor nodes, Node 1 and Node 2, and puts them on the open list. One successor node is generated for each value in d_1, the domain values of v_1 (PVAR-T1, the truck to deliver the package).

Node 1 is chosen from the open list for expansion because its value for zk, Truck-3, has a higher score (3) than Node 2's value of zk, Truck-1 (score = 2). Node 1's successors are

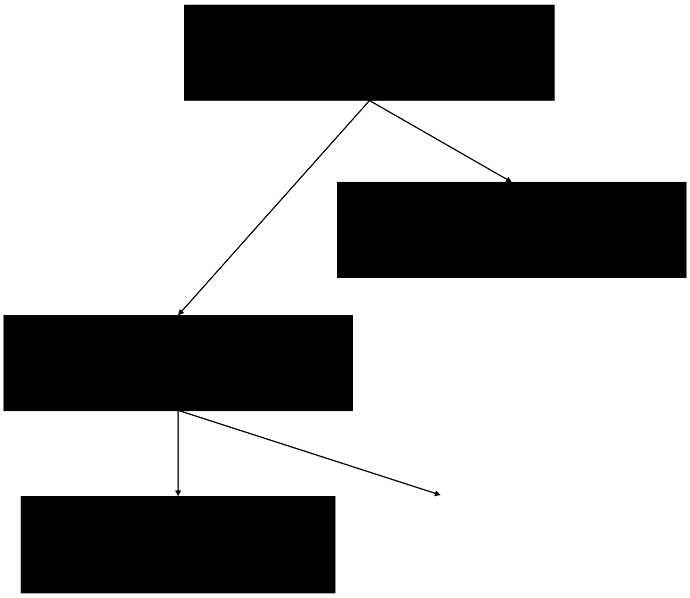

Figure 5.3: The top of the FCS search tree for Case-3

generated, one for each value in d_2, the domain of values for v_2 (PVAR-RT, the type of PVAR-T1). These include Node 4, for example, where zk = *Tanker-Truck*.

Procedure **check-forward** is invoked with argument $k = 2$.

Procedure **revise** is called with arguments $i = 2$ and $j = 1$. Procedure **check-vcons-binary** is invoked. It applies the hard binary constraint *hasType(PVAR-T1,PVAR-RT)* (from plan validation 1 of *Case-1*) The set of tuples $\langle c_1, c_2 \rangle$ (where c_k is a value for v_k) found to satisfy the constraint is: $\{\langle$Truck-3,Tanker-Truck$\rangle\}$. Thus d_1 gets the value $d_1 \cap$ {Truck-3} = {Truck-3} $\cap$ {Truck-3} = {Truck-3}. Arc consistency now holds between the v_1 (PVAR-T1) and v_2 (PVAR-RT) is obtained for the state represented by Node 1.

Procedure **revise** is next called with arguments $i = 3$ and $j = 1$ to obtain arc consistency between the v_1 (PVAR-T1) and v_3 (PVAR-L11C) in the state represented by Node 1. This is done by applying the soft binary constraint *at(PVAR-T1,PVAR-L11C)* (from plan validation 3 of *Case-1*).

Procedure **revise** is next called with arguments $i = 4$ and $j = 1$ to obtain arc consistency between the v_1 (PVAR-T1) and v_3 (PVAR-L11A) in the state represented by Node 1. This is done by applying the soft binary constraint *at(PVAR-T1,PVAR-L11A)* (from plan validation 2 of *Case-1*).

Node 2 is chosen for expansion next (as procedure **FCS** expands all nodes with $k = h$ before nodes with $k = h + 1$).

Procedure **check-forward** is invoked with argument $k = 2$.

Procedure **revise** is called with arguments $i = 2$ and $j = 1$. Procedure **check-vcons-binary** is invoked. It applies the hard binary constraint *hasType(PVAR-T1,PVAR-RT)* (from plan validation 1 of *Case-1*) The set of tuples $\langle c_1, c_2 \rangle$ found to satisfy the constraint is: $\{\langle$Truck-3,Tanker-Truck$\rangle\}$. Thus d_1 gets the value $d_1 \cap$ $\{$Truck-1$\}$ = $\{$Truck-3$\}$ $\cap$ $\{$Truck-1$\}$ = $\emptyset$. Thus arc consistency cannot be obtained between v_1 and v_2 if $v_1 = \{$Truck-1$\}$. In other words, *PVAR-T1* cannot be bound to *Truck-1* because then no value c can be found to bind to *PVAR-RT* such that c is a vehicle type compatible with the package type, *Liquid-Package*.

The procedure **fcs** continues and ultimately completes with the following result, after 9 nodes have been expanded, 93 nodes have been generated.

d_i	**Plan Variable**	**Source Value**	**Target Value**
1	PVAR-T1	Truck-1	Truck-3
2	PVAR-RT	Regular-Truck	Tanker-Truck
3	PVAR-L11C	City1-CL1	City1-CL1
4	PVAR-L11A	City1-CL1	City1-CL1

5.4 The Plan Instantiator

This section presents details about the Plan Instantiator, whose operation was illustrated in the example of Section 5.2. Once a global mapping has been produced by the Plan Mapper, it is used by the Plan Instantiator to instantiate a target plan from the chosen source plan. The target plan is a copy of the source plan with role fillers being replaced by constants from the target case or by newly-generated plan variables with values from the target case.

For example, suppose role r for action a_s has role value v_s, a plan variable, bound to constant c_s in the source plan. A target action a_t will be instantiated with role r's value v_t, a newly-generated plan variable, bound to case constant c_t in the source plan. The correspondence between c_s and c_t is indicated by the global mapping.

In addition to instantiating the new target plan (and its goals, subplans, actions, and validations), the Plan Instantiator creates a new HTN (Section 3.4.3) for the target plan and builds a temporary table of effects and a table of conditions for use by the Plan Modifier.

Given source plan p_s and the mapping created by the Plan Mapper, procedure **instantiate-plan** does the following to produce a new target plan.

1. Creates a new HTN h to hold the newly instantiated HTN nodes (plans and actions).

2. For each source plan variable v_s with value c_s, creates a new target plan variable v_t with value c_t, using the mapping c_s: c_t established by the Plan Mapper.

3. Reinstantiates *root* source plan p_s as *root* target plan p_t (via procedure **instantiate-plan1**(p_s, nil, h).

4. Updates the reductions for nodes in h.

5. Updates the validations for nodes in h. (Procedure **add-validation** is called on each validation in h.)

6. Updates the actions for plan nodes in h.

A plan is instantiated by procedure **instantiate-plan1**, called by procedure **instantiate-plan** with arguments source plan p_s, parent target plan p, and target HTN h. It does the following:

1. Creates an empty target plan p_t with p as its parent plan.

2. Adds p_t as a new node to N, the set of nodes in HTN h.

3. Sets the goals of p_t to be the reinstantiated goals of p_s (via procedure **instantiate-condition**)

4. Sets the actions of p_t to be the reinstantiated actions of p_s (via procedure **instantiate-action**)

5. Sets the validations of p_t to be the reinstantiated validations of p_s (via procedure **instantiate-validation**). Each validation v is added to the V, the set of validations in h.

6. Sets the subplans of p_t to be the reinstantiated subplans of p_s (via a recursive call to procedure **instantiate-plan1**).

A action is instantiated by procedure **instantiate-action1**, called by procedure **instantiate-plan1** with arguments source action a_s. parent target plan p, and target HTN h. It does the following:

1. Creates an empty target action a_t with p as its parent plan.

2. Adds a_t as a new node to N, the set of nodes in HTN h.

3. Sets the roles of a_t to be the reinstantiated roles of a_s.

4. Sets the validations of a_t to be the reinstantiated validations of a_s (via procedure **instantiate-validation**). Each validation v is added to the V, the set of validations in h.

Procedure **instantiate-validation** returns a copy of source validation v_s with the following:

- contributor(n) = analogue of contributor(v_s)

- consumer(n) = analogue of consumer(v_s)

- condition(n) = reinstantiated condition condition(v)

Procedure **instantiate-condition** returns a copy of source condition c_s with roles copied from c_s. Roles are copied using **instantiate-role** with returns a copy of source role r_s with roles fillers being replaced by the analogous target plan variables or constants.

The new HTN h created for target plan p_t by the above procedures has the following associated information:

- n_i, the new initial node (a dummy node that immediately precedes the first action in p_t)
- n_g, the new goal node (a dummy node that immediately follows the last action in p_t)
- n_r, the root node p_t
- N, the set of nodes of h. $N = R(P_t)$, the reduction of node p_t. Nodes correspond to e-actions or e-plans.
- V, the set of validations V of the nodes of h.

  $$V = \bigcup_{n \in N} V_n$$

  where V_n = e-preconditions(n) $\cup$ e-conditions(n) $\cup$ p-conditions(n).
- h_c, the Current Task Network (CTN) of h. The CTN contains nodes N_c = all $n \in N$ such that children(n) = $\emptyset$(i.e., all leaf nodes of h.)

5.5 Chapter Summary

This chapter discussed the following:

- CAPER reinstantiates a source plan for the target case using the abstracted framework of plan variables in the source plan created by CAPER and UM NONLIN.
- CAPER plan variables are reinstantiated in the target case by satisfying their plan variable constraints. These constraints come from the plan validations.
- Unary constraints are used to prune plan variable domains. Binary constraints are then solved using a forward-checking, constraint-satisfaction algorithm.
- Constraint checking involves additional querying of memory using probe graphs on CAPER's graph-structured KB.

Chapter 6

Adaptation in CAPER

Case-based planning systems requires effective and efficient plan adaptation mechanisms, as discussed in Section 2.4. The design approach in CAPER is to use flexible retrieval, supported by a graph-structured memory that is not preindexed, to find a source plan that closely matches that target problem. In the event that there are some differences between the source problem and target problem, the target plan (reinstantiated from a source plan by the process described in the previous chapter) will have to be adapted.

The goal of CAPER is to produce a plan that solves the target planning problem and that is correct with respect to CAPER's domain knowledge. Achieving this goal depends on the contents of the casebase. For example, if CAPER has never delivered a hazardous package before, it will not be able to adapt a source plan to deliver a target package that is hazardous. This is because CAPER is a reuse-only CBP system and thus relies extensively on the episodic knowledge of its casebase.

CAPER's adaptation methods are currently only used to adapt a plan prior to its execution. They are *not* used to repair plan failures found during execution, as CHEF (Hammond, 1986), for example, can do. CHEF is required to handle plan failures because its adaptor component lacks a complete causal domain model.[1] Much research has been done on diagnosing and repairing plan failures, including those in plans produced by CBP systems (e.g., (Alterman, 1988; McCartney, 1993)). To repair plan failures, CAPER would need mechanisms for getting feedback from plan execution and for diagnosing plan execution failures.

Many of CAPER's adaptation methods are based on those used in PRIAR (Kambhampati, 1989). Like PRIAR, CAPER makes use of plan validations (Section 3.4.6), which capture plan dependencies. CAPER uses modified versions of PRIAR's algorithms for annotation verification and refit task creation. Annotation verification is the process by which a validation structure is examined in the target case to find *applicability* failures, i.e., those resulting from differences between the source and target cases. The term "failure" is used here in a neutral sense to mean any mismatch, favorable or not, between the source and

---

[1] Such a model, however, is present in CHEF's simulator *and* explainer so that it is possible for CHEF to always diagnose the cause of a plan execution failure. It is unclear why this domain knowledge is not made available to the adaptor, instead of the explainer, so that it could predict such failures before plan execution.

target cases that causes inconsistencies in the validation structure of the target plan.

Refit tasks are created to repair applicability failures. In PRIAR, refit tasks include achieving a condition by reducing a goal task, rereducing a goal task using an alternate method, and pruning subplans, actions, and validations from the plan.[2] PRIAR uses a generative planner based on NONLIN (Tate, 1977) to perform some refit tasks. In this case, the refit task dictates modifications to the target plan's HTN, which is then passed to the generative planner. The generative planner expands the task network into a fully-specified plan.

PRIAR's approach to plan adaptation requires a full generative planner and complete causal domain model. CAPER does not have a full generative planner available to it (although this is a possible extension to CAPER see Section 9.2). It also requires less explicit causal domain knowledge (see Section 6.4).

In brief, adaptation in CAPER involves the following modules:

- The **Plan Checker** (Section 6.1) takes a target plan as input and checks its validations for applicability failures. It does this by annotation verification and other methods. It produces a set of plan modification tasks.

- The **Plan Modifier** (Section 6.2) takes a target plan and a set of plan modification tasks as input. It modifies the target plan to achieve these tasks. It calls the Plan Retriever to retrieve substitute or patch plans. The modified plan is the checked by the Plan Checker, and the process is repeated if necessary.

- The **Plan Merger** (Section 6.3) takes two target plans and merges them. Harmful interactions that result from this are detected (by the Plan Checker) and repaired (by the Plan Modifier).

- The **Plan Storer** (Section 6.5) takes the final (successful) plan, after all plan modification tasks have been performed, and stores it, along with its problem specification, as a new case in the KB. Temporary working data structures describing the plan are converted to KB frames.

The example presented in the previous chapter is continued in this chapter (in Section 6.1.1 and Section 6.2.1) to illustrate the plan adaptation process. Then the modules involved are described in turn. Section 6.4 contains some discussion of additional plan adaptation issues in CAPER.

6.1 Plan Checker

This section describes the Plan Checker. An example is presented followed by additional details on the Plan Checker modules.

---

[2] "Reducing" a goal task is done by finding a schema the creates a subplan for achieving the goal.

6.1.1 An Example

This section continues the UM TRANSLOG domain example of Section 5.2, in which the target problem of Case-3 was to deliver a non-hazardous, non-valuable liquid package, *Pkg-3.1*, from *City1-CL1* to *City1-CL2*. A target plan, Plan-3, has been instantiated for the Case-3 from source plan *Plan-Located-At-1.2* (Plan-1) of Case-1. The yet unadapted Plan-3 currently has the following actions:

```
 1. Do-Nothing-3.1
 2. Do-Nothing-3.2
 3. P-Collect-Fees-3.1
       package: PVAR3-PKG (Pkg-3.1)
 4. Do-Nothing-3.3
 5. P-Open-Door-3.1
       vehicle: PVAR3-VEH (Truck-3)
 6. P-Load-Package-3.1
       package: PVAR3-PKG (Pkg-3.1)
       vehicle: PVAR3-VEH (Truck-3)
 7. P-Close-Door-3.1
       vehicle: PVAR3-VEH (Truck-3)
 8. P-Move-Vehicle-3.1
       vehicle:     PVAR3-VEH (Truck-3)
       origin:      PVAR-LOCA (City1-Cl1)
       destination: PVAR-DEST (City1-Cl2)
       route:       P-Local-Road-Route
 9. P-Open-Door-3.2
       vehicle: PVAR3-VEH (Truck-3)
10. P-Unload-Package-3.1:
       package: PVAR3-PKG (Pkg-3.1)
       vehicle: PVAR3-VEH (Truck-3)
11. P-Close-Door-3.2
       vehicle: PVAR3-VEH (Truck-3)
12. Do-Nothing-3.4
13. P-Deliver-3.1
       package: PVAR3-PKG (Pkg-3.1)
```

The task of the Plan Checker is to check this plan to predict if it will work in the target case. Typically some adaptation will have to be done to the plan due to differences between a source and target cases. These differences are detected by the Plan Checker by a process called **annotation verification**, based on that of PRIAR (Kambhampati and Hendler, 1992).

The following are some of the possible differences between a source and target UM TRANSLOG case and the corresponding modifications to the plan that may need to be made:

- If the source package and analogous target package have different types, a different type vehicle may be required in the target plan.

- If the source package origin (destination) does not match the target package origin (destination), additional actions may need to be added to the plan or substitute routes (and transport centers) found.
- If the source package is hazardous and/or valuable and the target package is not (or vice versa), origin/destination, additional actions need to be added to the plan. In addition, hazardous packages must not be routed through cities where they are forbidden.
- If the vehicle or transport center used in the source plan is unavailable in the target case, a substitute will have to be found. For example, a truck may be in the shop for repair or the airport closed due to bad weather.
- If the source vehicle and analogous target vehicle have different types, a different subplan for loading (unloading) the vehicle and/or a different route (and transport centers) may be required.

Some of these differences may not be relevant, e.g., if Case-X uses a blue truck to transport the package and Case-Y uses a red truck. This difference is not relevant because color of the vehicle does not impact the plan in the current UM TRANSLOG domain model. Thus CAPER needs to determine the *relevant* differences between an old and new case.

CAPER does this using the plan validations and problem descriptions of the source and target cases. Plan validations capture dependencies among plan actions and the initial situation. Using the validations, CAPER can determine which facts of the source problem's initial situation were relevant to the source plan.

The source plan Plan-1, for example, depends on the fact that *Truck-1* is at location *City1-CL1*, the origin of *Pkg-1.1*, in the initial situation of Case-1, Case-Make-Deliv-1-InitSit (Sit-1). This dependency is captured in validation 14 with contributor Sit-1. This validation has condition *at(PVAR-T1,PVAR-L11B)*, where plan variables *PVAR-T1* and *PVAR-L11B* are bound to truck *Truck-1* and package origin *City1-CL1*, respectively. The ground validation condition matches the initial state fact *at(Truck-1,*City1-CL1*)*. The complete list of validations for Plan-1 is shown in Appendix C.2.3.

The Plan Instantiator reinstantiates validation 14 in Case-3 to have contributor Case-Make-Deliv-3-InitSit and condition *at(PVAR3-VEH,PVAR3-ORGN)*, where plan variables *PVAR-ORGN* and *PVAR-L11B* are bound to truck *Truck-3* and package origin *City1-CL1*. A condition is successfully matched to a fact of the initial situation via a PARKA query.

Thus there are several uses for plan validations in CAPER. Recall that validations were also used by the Plan Mapper to constrain the rebinding of *PVAR-T1* and hence map *Truck-1* in the source case to *Truck-3* in the target case.

6.1.2 A Hypothetical Fault

Suppose that in Case-3's initial situation, *Truck-3* were *not* at *City1-CL1*. In that case, Plan-3 would not work because *Truck-3* needs to be at the the location of *Pkg-3.1* in order to load the package (via action P-Load-Package-3.1). CAPER would detect this **applicability**

failure of the Plan-3 by noticing that validation 14 does not hold, i.e. its ground condition *at(Truck-3,*City1-CL1*)* would *not* match any fact in the initial situation. Validation 14 is thus a **missing validation** and the fact *at(Truck-3,*City1-CL1*)* is thus an **out fact**. (See Section 6.1.4 for details.)

The condition *at(Truck-3,City1-CL1)* is achievable: *Truck-3* could be moved to *City1-CL1* from *City1-CL2* by an instance of the action *P-Move-Vehicle*. When a Plan Modifier detects a missing validation, it creates a **plan modification task** to achieve that validation's condition.

The Plan Modifier calls upon Plan Retriever to find a plan that achieves this condition as its goal. (See Section 6.2.4 for details.) Suppose a one-action source plan is retrieved and its action mapped to the target case. This action, P-Move-Vehicle-3.0 (move *Truck-3* to *City1-CL1* from *City1-CL2*), would need to be **merged** into the target plan by the Plan Merger (Section 6.3).

Introducing this action, however, might cause a harmful **interaction**. For example, suppose *Truck-3* is to carry an additional (liquid-type) package, Pkg-3.2, initially located at *City1-CL2*. If Pkg-3.2 is also to be carried by *Truck-3*, it must be loaded at *City1-CL2*.

The initial situation is the following:

City1-CL1	**City1-CL2**
Pkg-3.1	*Pkg-3.2* *Truck-3*

The three initial actions, in no order, are:

- P-Load-Package-3.1 which loads *Pkg-3.1* into *Truck-3* at *City1-CL1*
- P-Load-Package-3.2 which loads *Pkg-3.2* into *Truck-3* at *City1-CL2*
- P-Move-Vehicle-3.0 which moves *Truck-3* from *City1-CL2* to *City1-CL1*

P-Move-Vehicle-3.0 must precede P-Load-Package-3.1.

The action P-Load-Package-3.1 has a precondition *at(Truck-3,City1-CL1)*. This precondition is stored as the condition of validation 14, with contributor Case-Make-Deliv-3-InitSit (Sit-3). Assume the new action, P-Move-Vehicle-3.0, comes between the initial situation and P-Load-Package-3.1, i.e.,

- P-Move-Vehicle-3.0 which moves *Truck-3* from *City1-CL2* to *City1-CL1*
- P-Load-Package-3.1 which loads *Pkg-3.1* into *Truck-3* at *City1-CL1*
- P-Load-Package-3.2 which loads *Pkg-3.2* into *Truck-3* at *City1-CL2*

Then the condition will be clobbered and validation 14 will fail. In other words, *Truck-3* is being moved from *City1-CL2* to *City1-CL1* before Pkg-3.1 has been loaded onto it.

This is known as an **interaction**. CAPER must detect a failing validation caused by introducing a new action in the plan.

The Plan Modifier then must correct this interaction, in this case by putting P-Move-Vehicle-3.0 *after* P-Load-Package-3.1 in the plan. Thus first Pkg-3.2 will be loaded into *Truck-3* at *City1-CL2* then *Truck-3* will be driven to *City1-CL1*, where *Pkg-3.1* can be loaded. Introducing an ordering between two actions in the plan is called **linearization**.

The resulting action order is:

1. P-Load-Package-3.2 which loads Pkg-3.2 into *Truck-3* at *City1-CL2*
2. P-Move-Vehicle-3.0 which moves *Truck-3* from *City1-CL2* to *City1-CL1*
3. P-Load-Package-3.1 which loads *Pkg-3.1* into *Truck-3* at *City1-CL1*

6.1.3 An Actual Fault

The main difference between the goal of Case-1 and the goal of Case-3 is that in the former, the package *Pkg-1.1* is a regular-type package and in the latter, the package *Pkg-3.1* is a liquid-type package. As seen in the Section 5.2, this difference causes *Truck-3*, a tanker truck, to be selected over *Truck-1*, a regular package.

In Plan-1, the subplan *Pkg-Loaded-1.2* was used for loading *Truck-1*. It contained the actions P-Open-Door, P-Load-Package, and P-Close-Door. This subplan will *not* work in Plan-3 for loading *Truck-3* because the loading of a tanker truck requires the actions P-Connect-Hose, P-Open-Valve, P-Fill-Tank, P-Close-Valve, and P-Disconnect-Hose.

Again, a plan validation is used to detect this applicability failure. As shown in Figure 6.1, subplan *Pkg-Loaded-3.2*, the reinstantiation of *Pkg-Loaded-1.2* in Case-3, has e-precondition validation 22 with (ground) condition *hasType(Truck-3,Regular-Truck)* and contributor Case-Make-Deliv-3-InitSit. This condition does not match any fact in Case-3's initial situation, because that situation asserts fact *hasType(Truck-3,Tanker-Truck)* and a truck can only have a single (physical) subtype (e.g., tanker, regular, etc.). The condition is *not* achievable by the planner as no action (in the domain model) can change the type of *Truck-3*.

In this event, the need for this validation condition to hold must be eliminated. This validation condition was originally introduced into Plan-1 by the planner by applying the regular-vehicle load method to generate subplan *Pkg-Loaded-1.2*. The goal of subplan *Pkg-Loaded-3.2* is to load the package into the vehicle, i.e., *at(Pkg-3.1,Truck-3)*. CAPER must find an alternate subplan that achieves this goal but which does *not* require the unachievable condition *hasType(Truck-3,Regular-Truck)* to hold.

CAPER does this by pruning the subplan *Pkg-Loaded-3.2* and **rereducing**— finding an alternate method for achieving — its goal of *at(Pkg-3.1,Truck-3)*. This example will be continued after a description of the annotation verification process.

6.1.4 Annotation Verification

The annotation verification procedures checks the target plan's (or subplan's) validation structure and creates new plan modification tasks to handle applicability failure arising

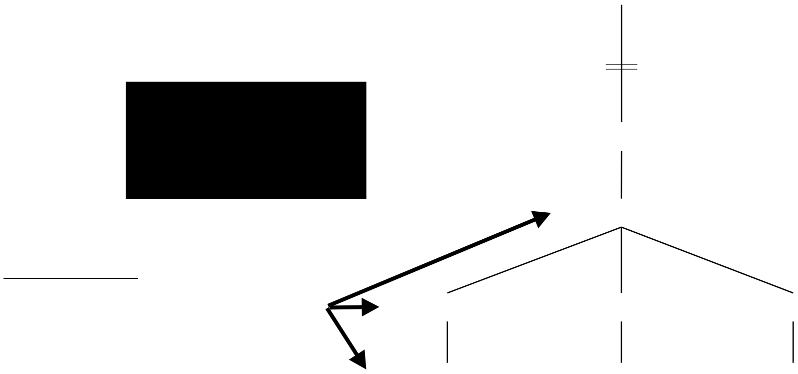

Figure 6.1: Failing filter-condition hasType(Truck-3,Tanker-Truck) of validation 22 in Plan-3

from differences between the target problem and the corresponding source problem. These routines are largely based on algorithms in PRIAR, but they have been modified to work with the data structures in the working data and the KB of CAPER. Here I present the theory behind these algorithms only briefly, simplifying some of the terminology for clearer exposition. More detailed information can be found in (Kambhampati, 1989).

Given a fact f_s in the source problem, f_m is the mapped fact for f_s. In other words, any source variables in f_s have been replaced by their analogous target variables as dictated by the mapping found by the Plan Mapper. For example, if source plan variable X maps to target plan variable Y and $f_s = p(X)$, then $f_m = p(Y)$. The source plan initial situation I_s is mapped to situation I_m by putting mapped fact f_m into I_m for every fact f_s in I_s.

Similarly, g_m is the mapped goal for $g_s \in G_s$, the goals of the source problem. G_m is the set of mapped goals. G_t are the goals of the target problem. I_t is the target situation.

The following differences, to borrow PRIAR's terminology, may exist:

- An **out fact** is a fact $f_m \in I_m$ and $\neg(f_m \in I_t)$.
- A **new fact** is a fact $f_t \in I_t$ and $\neg\exists f_m$ such that $f_m = f_t$.
- An **extra goal** is a goal $g_m \in G_m$ and $\neg(g_m \in G_t)$.
- An **unnecessary goal** is a goal $g_t \in G_t$ and $\neg\exists g_m$ such that $g_m = g_t$.

These differences can cause applicability failures in the target plan.

Given an **unnecessary goal** g, the part of the plan achieving g may be prunable. This will occur later on if the subplan for g does not serve any other purpose besides establishing g. All validations v with condition(v) = g and consumer(v) = n_g, the goal situation of the target plan, are classified as **unnecessary validations** in CAPER (and PRIAR). Thus they are scheduled for pruning (a prune plan modification task is created for each).

An **extra goal** g must be established by augmenting the target plan with a new subplan that achieves g. PRIAR creates a new subplan for g from scratch using its generative planner (NONLIN). CAPER, which lacks a generative planner component, will retrieve a source plan from memory to adapt into a target (sub)plan for achieving g. A validation v is created with condition(v) $= g$ and consumer(v) $= n_g$. This is termed a **missing validation**, because it has yet to be established in the source case (i.e., it lacks a contributor).

An **out fact** f has a corresponding validation v with condition(v) $= f$ and no contributor. Validation v is termed a **failing validation** because it held in the source case but does not hold in the target case. If condition(v) is a precondition, it must be reestablished in a manner similar to that of an extra goal. If condition(v) is a filter condition, it must be made superfluous (and thus prunable) by replacing the subplan p that requires it with an alternate plan that does not. This is done by **rereducing** a node in the plan, which involves retrieving a *replacement* plan which lacks the offending filter condition.

A **new fact** f_t may be serendipitous if there exists some validation v in the target plan such that condition(v) $= f_t$. The target subplan that supplies v may now be unnecessary and hence prunable. By pruning an unnecessary subplan, the overall length of the target plan will be reduced. CAPER does not currently take advantage of new facts.

The applicability failures of missing, failing, and unnecessary validations are detected by procedure **annotation-verification**. Repair routines are invoked to create plan modification tasks for processing by the Plan Modifier. Procedure **annotation-verification** is invoked by the Plan Checker's main routine **check-plan** with argument target plan p_t and does the following:[3]

1. For each goal g in G_u, the set of unnecessary goals,
    (a) For each validation v in $A_p(n_g)$ with condition $= n_g$, create a plan modification of type *prune* for plan p and validation v. $A_p(n) =$ e-preconditions(n) $\cup$ p-conditions(n).
2. For each goal g in G_e, the set of extra goals, call **repair-missing-validation**(v).
3. For each v in V_o, the set of out fact validations,
    - If condition(v) has type *precondition*, call **repair-failing-precondition-validation**(v).
    - If condition(v) has type *filter-condition*, call **repair-failing-filter-condition-validation**(v).
    - If condition(v) has type *phantom-condition*, call **repair-failing-phantom-condition-validation**(v).

---

[3] This procedure is based on that of PRIAR.

(See Section 3.4.4 for a description of these condition types and Section 6.2.5 for a description of these procedures.)

Unnecessary goals G_u are computed by examining each goal g_m that has been mapped from the source plan during the elaboration phase. If g_m does not correspond to a goal feature in the case features of the target problem description, then $g_m \in G_u$.

Extra goals G_e are computed by examining each case feature g for a goal of the target plan. If g was not mapped to any source plan goal feature during plan retrieval, it is returned as an extra goal.

Out fact validations are computed by checking each validation of p_t with contributor(v) = n_i. To check v, its condition (already mapped to the target case during the elaboration phase) is evaluated in the target initial situation via a PARKA query. The validations whose query fails are failing validations and their conditions are out facts.

In the ongoing example of this chapter (Section 6.2.1), the above procedure is executed on Plan-3 as follows:

1. Plan-3 is checked for unnecessary goals. There is only one goal from the source plan, *Case-Make-Deliv-1-Goal-1*, and it was mapped to target goal *Case-Make-Deliv-3-Goal-1*. Thus there are no unnecessary goals.
2. Plan-3 is checked for extra goals. There is only one goal feature (*Feature-3.1*) for Case-3, and it was mapped to source plan goal feature (*Feature-3.1s*). Thus there are no extra goals.
3. Plan-3 is checked for out facts. Validations whose contributor is n_i (*Case-Make-Deliv-3-InitSit*) are checked by issuing PARKA queries. Two such validation have conditions not holding:
    - Validation 22 — condition: filter-condition *hasType(PVAR3-VEH,Regular-Truck)*, consumer: action *P-Open-Door-3.1*
    - Validation 28 — condition: filter-condition *hasType(PVAR3-VEH,Regular-Truck)*, consumer: action *P-Open-Door-3.2*

   As both of these validations have failing filter conditions, procedure **repair-failing-filter-condition-validation** is called for each, as described in Section 6.2.1.

6.2 Plan Modifier

This section describes the Plan Modifier. An example is presented followed by additional details on the Plan Modifier modules.

6.2.1 Example

This section continues with the running example of this chapter. The Plan Checker has detected the following two out fact validations by annotation verification (Section 6.1.4):

- Validation 22 — condition: filter-condition *hasType(PVAR3-VEH,Regular-Truck)*, consumer: action *P-Open-Door-3.1*

- Validation 28 — condition: filter-condition *hasType(PVAR3-VEH,Regular-Truck)*, consumer: action *P-Open-Door-3.2*

Both of these conditions fail because *Truck-3*, the value of plan variable *PVAR3-VEH*, has type *Tanker-Truck*, rather than type *Regular-Truck* (as *Truck-1* had in Plan-1). The type of *Truck-3* is a filter condition, because it cannot be changed by the planner.

As discussed in Section 6.1.1, CAPER handles failing filter condition validations by replacing the subplan that causes the failing condition to be needed with an alternate subplan that achieves the same goal.

CAPER determines that the condition of validation 22 was introduced into the original plan for subplan *Pkg-Loaded-1.2*, which has been reinstantiated as subplan *Pkg-Loaded-3.2* in the target plan.

Thus CAPER has to find an alternate (sub)plan that achieves the goal of *Pkg-Loaded-3.2* which is *at(Pkg-3.1,Truck-3)*, i.e., the package loaded onto *Truck-3*.

CAPER finds a **replacement plan** for rereducing a goal, a **subproblem** of the target problem, by invoking the Plan Retriever with a retrieval description containing a feature that corresponds to the goal to be rereduced. (See Section 6.2.7 for details.)

In this example, CAPER must find an alternate plan for loading *Pkg-3.1* onto *Truck-3*, a tanker truck. The Plan Retriever attempts to find a plan for loading a package onto *Truck-3*. An initial retrieval description is generated and "executed". No plan is matched in the KB, and facet *Truck-3* is generalized from the constant *Truck-3* to the variable t, where t has type *Tanker-Truck*.

CAPER goes back to memory and retrieves a (sub)plan. The new retrieval description succeeds: a single source plan, *Pkg-Loaded-2.1* ("Plan-2a"), is matched. Plan-2a is a plan for loading package *Pkg-2.1* onto tanker truck Truck-4. Plan-2a is a subplan of plan *Plan-Located-At-2.4*, the plan for *Case-Make-Deliveries-2* ("Case-2"). In that case, a liquid package *Pkg-2.1* was delivered from *City1-CL2* to *City2-CL1* via tanker truck *Truck-4*.

Plan-2a has the following actions:

```
1. P-Connect-Hose-2.1
      package: Pkg-2.1
      vehicle: Truck-4
2. P-Open-Valve-2.1
      vehicle: Truck-4
3. P-Fill-Tank-2.1
      vehicle: Truck-4
4. P-Close-Valve-2.1
      vehicle: Truck-4
5. P-Disconnect-Hose-2.1
      package: Pkg-2.1
      vehicle: Truck-4
```

Plan-2a is then given to the Plan Mapper and Plan Instantiator which instantiate a new target plan, Plan-3a, for loading *Pkg-3.1* onto *Truck-3*. Plan-3a has the following actions:

```
 1. P-Connect-Hose-3.1
      package: Pkg-3.1
      vehicle: Truck-3
 2. P-Open-Valve-3.1
      vehicle: Truck-3
 3. P-Fill-Tank-3.1
      vehicle: Truck-3
 4. P-Close-Valve-3.1
      vehicle: Truck-3
 5. P-Disconnect-Hose-3.1
      package: Pkg-3.1
      vehicle: Truck-3
```

As shown in Figure 6.2, the newly instantiated plan Plan-3a must now be merged into Plan-3, replacing *Plan-Pkg-Loaded-3.2* as a subplan of *Pkg-Loaded-Val-3.1*.

The old plan's subplans, actions, and any of validations that are now unnecessary are pruned from Plan-3. In addition to checking for unnecessary validations, the Plan Checker also checks for missing and failing validations (including plan interactions) as a result of merging Plan-3a into Plan-3. In this case, the only plan modification tasks that need to be performed involve pruning now unnecessary validations.

After merging Plan-3a and Plan-3, Plan-3 has the following actions:

```
 1. Do-Nothing-3.1
 2. Do-Nothing-3.2
 3. P-Collect-Fees-3.1
      package: PVAR3-PKG (Pkg-3.1)
 4. Do-Nothing-3.3
 5. P-Connect-Hose-3.1
      package: Pkg-3.1
      vehicle: Truck-3
 6. P-Open-Valve-3.1
      vehicle: Truck-3
 7. P-Fill-Tank-3.1
      vehicle: Truck-3
 8. P-Close-Valve-3.1
      vehicle: Truck-3
 9. P-Disconnect-Hose-3.1
      package: Pkg-3.1
      vehicle: Truck-3
10. P-Move-Vehicle-3.1
      vehicle:     PVAR3-VEH (Truck-3)
      origin:      PVAR-LOCA (City1-Cl1)
      destination: PVAR-DEST (City1-Cl2)
      route:       P-Local-Road-Route
11. P-Open-Door-3.2
```

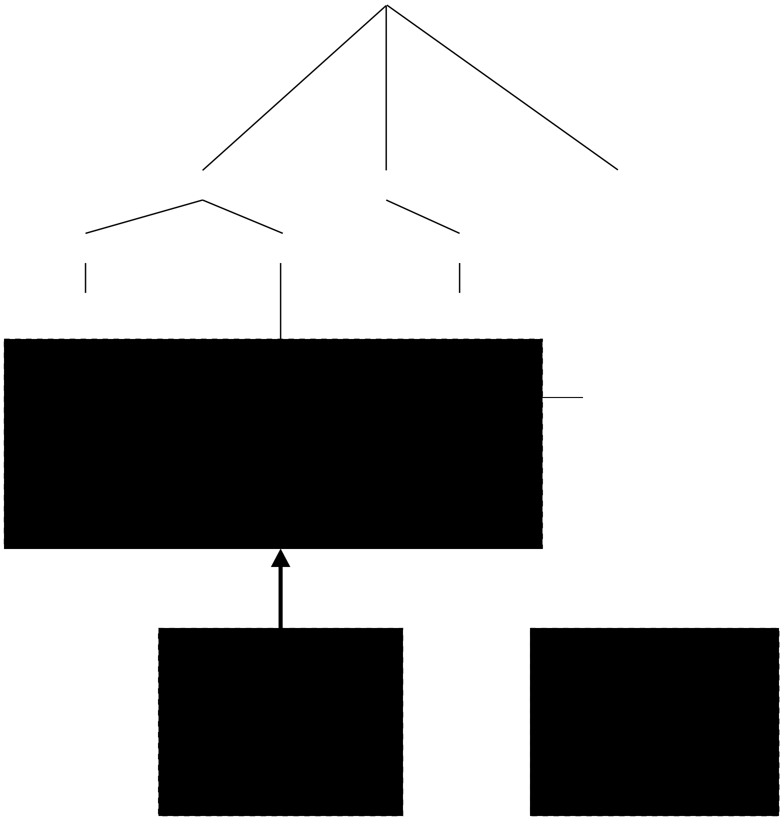

Figure 6.2: The replacement of a target subplan by Plan-3a, a reinstantiated of Plan-2b.

```
       vehicle: PVAR3-VEH (Truck-3)
12. P-Unload-Package-3.1:
       package: PVAR3-PKG (Pkg-3.1)
       vehicle: PVAR3-VEH (Truck-3)
13. P-Close-Door-3.2
       vehicle: PVAR3-VEH (Truck-3)
14. Do-Nothing-3.4
15. P-Deliver-3.1
       package: PVAR3-PKG (Pkg-3.1)
```

By a process very similar to the above, the out fact for validation 28 is also repaired. Plan-3b, a plan for unloading *Pkg-3.1* from *Truck-3*, is instantiated from *Not-Pkg-Loaded-2.1* ("Plan-2b"), the subplan from Case-3 for unloading package *Pkg-2.1* from tanker truck *Truck-4*. Plan-2b has the following actions:

```
 1. P-Connect-Hose-2.2
       package: Pkg-2.1
       vehicle: Truck-4
 2. P-Open-Valve-2.2
       vehicle: Truck-4
 3. P-Fill-Tank-2.1
       vehicle: Truck-4
 4. P-Close-Valve-2.2
       vehicle: Truck-4
 5. P-Disconnect-Hose-2.2
       package: Pkg-2.1
       vehicle: Truck-4
```

After merging Plan-3b and Plan-3, Plan-3 has the following actions:

```
 1. Do-Nothing-3.1
 2. Do-Nothing-3.2
 3. P-Collect-Fees-3.1
       package: PVAR3-PKG (Pkg-3.1)
 4. Do-Nothing-3.3
 5. P-Connect-Hose-3.1
       package: Pkg-3.1
       vehicle: Truck-3
 6. P-Open-Valve-3.1
       vehicle: Truck-3
 7. P-Fill-Tank-3.1
       vehicle: Truck-3
 8. P-Close-Valve-3.1
       vehicle: Truck-3
 9. P-Disconnect-Hose-3.1
       package: Pkg-3.1
       vehicle: Truck-3
10. P-Move-Vehicle-3.1
       vehicle:     PVAR3-VEH (Truck-3)
       origin:      PVAR-LOCA (City1-Cl1)
       destination: PVAR-DEST (City1-Cl2)
       route:       P-Local-Road-Route
11. P-Connect-Hose-3.2
       package: Pkg-3.1
       vehicle: Truck-3
12. P-Open-Valve-3.2
       vehicle: Truck-3
13. P-Empty-Tank-3.1
       vehicle: Truck-3
14. P-Close-Valve-3.2
       vehicle: Truck-3
15. P-Disconnect-Hose-3.2
       package: Pkg-3.1
       vehicle: Truck-3
16. Do-Nothing-3.4
```

```
17. P-Deliver-3.1
      package: PVAR3-PKG (Pkg-3.1)
```

The complete, correct Plan-3 is now stored by the Plan Storer (Section 6.5) in the KB as plan *Pkg-Located-At-3.2*, the plan for case *Case-Make-Deliv-3*.

6.2.2 Main Procedure

Procedure **modify-plan**, the main Plan Modifier procedure, is invoked by CAPER's Main Module. It processes those plan modification tasks that have been put on the plan modification task queue by Plan Checker procedures (or by other Plan Modifier procedures). Processing a task may cause additional tasks to be put on the queue. Tasks are processed until the queue is empty.

This procedure does the following:

1. If the plan modification task queue is empty, return from **modify-plan**.
2. Remove plan modification task t for plan p from the plan modification task queue.
3. Compute M, the set of applicable plan modifications for task t.
4. Choose a plan modification m in M.
5. Apply the plan modification m to plan p.
6. If m is successful, goto Step 1.
7. If m is unsuccessful, goto Step 5.

When a new plan modification task is added to the queue, an insertion sort of the queue is performed. Currently tasks are ordered by their general type: rereduce tasks are ordered before establish tasks which are ordered before prune tasks. A similar scheme was used in PRIAR, although it processed rereduce tasks immediately upon their creation. Pruning of actions is done last in the event that the action might be reusable for some modification of the plan (e.g., establishing a failing validation).

An enhancement to this procedure would be to combine two or more plan modification tasks in the queue. For example, suppose there are two plan modification tasks: achieve goal g_1 and achieve goal g_2. These could be combined into a single task "achieve $g_1 \wedge g_2$" for which a single source plan might be retrieved.

One or more generic plan modifications are associated with each type of plan modification task.

6.2.3 Handling Unnecessary Validations

Procedure **prune-validation** is invoked by the Plan Modifier for tasks of type *prune*. It takes v, an unnecessary validation, as its argument. As described in Section 6.1.4, unnecessary validations are those whose consumers no longer exist or whose conditions have been satisfied by another contributor (including the target problem's initial situation). This procedure is based on that of PRIAR. Details can be found in (Kambhampati, 1989)[p. 45]. In brief, the procedure prunes v by removing it from V, the set of validations in the target plan's HTN.

If n was the contributor of v, it may be possible to prune additional nodes (actions and plans) from the HTN that are no longer serving any purpose. CAPER currently does *not* prune these nodes because it is possible that a node n that is supplying no validations (i.e., e-conditions(n) $= \emptyset$) is actually serving a purpose. In HTN planning, it is possible for a node n in the HTN to be expanded into nodes n_1, n_2,. . . ,n_r by some method where a node n_i has no effects. Thus a node n_i cannot be a contributor and therefore has no e-conditions.

For example, consider the method for expanding a goal node g: "go to the store and come back". A method m ("make a round trip") is used and g is expanded into action nodes: (1) go to the store from l (the current location) and (2) go to l from the store. Because the final state after action (2) is the same as the initial state (i.e., the agent is at l), node g has no effects and hence no e-conditions. Pruning g from the plan, however, would be a mistake. The purpose of g is dictated by method m, not g's nonexistent e-conditions.

The ability to have tasks without effects allows greater expressivity when encoding a planning domain (Erol, 1995). PRIAR's encoding of the Blocks World domain, however, did not make use of this kind of encoding, i.e., every action put-on(x, y) has effects (e.g., on(x, y), etc.). Thus PRIAR knows that a node with no e-conditions is serving no purpose in the plan and hence can safely be pruned without leaving an incorrect plan.

6.2.4 Handling Missing Validations

Procedure **repair-missing-validation**, which handles missing validations, is invoked by the Plan Checker immediately after a missing validation is detected during annotation verification. Its takes v, a missing validation, as its argument. As described in Section 6.1.4, a missing validation v is one whose condition c has not been established for v's consumer n. This is a result of an extra goal $g = c$. These procedures are based on that of PRIAR. Details can be found in (Kambhampati, 1989)[p. 47].

Procedure **repair-missing-validation** creates a plan modification task of type *achieve* with condition c, validation v and consumer n_g. This task will be processed (see Section 6.2.6) and an alternate contributor will be found for c.

6.2.5 Handling Failing Validations

Several procedures for handling missing validations are invoked by the Plan Checker immediately after a failing validation is detected during annotation verification. As described in Section 6.1.4, a failing validation v is one currently lacking a contributor in the target plan (including the target initial situation). These procedures are based on that of PRIAR. Details can be found in (Kambhampati, 1989)[pp. 49-51]. These procedures cause plan modification tasks to be created to establish c, the condition of v. The particular missing validation repair procedure called depends on whether c is a precondition, filter condition, or phantom condition. (See Section 3.4.4 for a description of these condition types.)

Handling Failing Precondition Validations

Because preconditions are achievable by the planner, procedure **repair-failing-precondition-validation** creates a plan modification task of type *achieve* with condition c, validation v and consumer n_d, the consumer of v. This task will be processed (see Section 6.2.6) and an alternate contributor will be found for c.

Handling Failing Filter Condition Validations

Because filter conditions are *not* achievable by the planner, procedure **repair-failing-precondition-validation** creates a plan modification task of type *rereduce* for some node n such that v's consumer $n_d \in R(n)$ and $v \in$ e-conditions(n) and $\neg$ ($v \in$ e-conditions(m)), where m is the parent of n. The current nodes in $R(n)$ are pruned from the HTN. This task will be processed (see Section 6.2.7). An example of this was given in Section 6.2.1 for validation 22.

Handling Failing Phantom Condition Validations

Phantom conditions are ones that did not have to be achieved by the planner because they were already established. A failing phantom condition is one that is no longer established. Thus it must be converted from a phantom condition back to its original condition type of *precondition* and processed by the planner. A plan modification task is created of type *dephantomize* for condition v with consumer n_d the consumer of v. This task will be processed by the Plan Modifier procedure **pmod-rereduce** (Section 6.2.7).

For example, suppose package P and Truck T are initially at location L in the source case. Thus action P-Load-Truck(P,T) has its precondition C, at(T,L), already satisfied. The planner would convert this precondition to a phantom condition and not have to achieve it. Now suppose C does not hold in the target case (i.e., T is not at L). Condition C will have to be dephantomized and then, as a precondition, be achieved by the planner. A (sub)plan will have to be found to achieve C (i.e., move T to L) and the subplan (mapped and instantiated for the target case) will have to be merged into the target plan.

6.2.6 Establishing Conditions

Failing or missing validations need to have their conditions established in the target plan. A condition is to be established at a particular node, i.e., the validation's consumer. CAPER first tries to establish a condition finding an existing contributor for it in the current plan.

It does this by calling a **Q&A** procedure similar to that of partial-order planners such as UM NONLIN (Ghosh *et al.*, 1992).

Suppose that CAPER is attempting to establish condition c at node n. As described in Section 5.4, CAPER builds a table of known effects. Each node in a target HTN has its effects recorded in this table. Note that these effects have been instantiated with actual plan variables. This table of effects is used to determine the potential contributors and clobberers for c.

Assume that c is to be true at n. An analogous procedure is used if c is to be false at n. CAPER determines the status of c at n as follows:

1. It computes E, the set of nodes that have an effect e such that e unifies with c.
2. It computes A, the set of nodes in E asserting e.
3. It computes D, the set of nodes in E deleting e.
4. It computes A_b and A_p, the sets of nodes in A that come before (directly or indirectly) n and that are in parallel with n, respectively. (A node can be in parallel with n given that the set of nodes is partial order.)
5. It computes D_b and D_p, the sets of nodes in D that come before (directly or indirectly) n and that are in parallel with n, respectively.
6. Let $M = A_b \cup A_p \cup D_b \cup D_p$.
7. For each node $m \in M$ such that no node q precedes m, where $q \in M$, put m in M'.
8. Let $A_b = A_b \cup M'$.
9. Let $A_p = A_p \cup M'$
10. Let $D_b = D_b \cup M'$.
11. Let $D_p = D_p \cup M'$.

The status of c at n is:

1. c is necessarily true at n if $A_b \neq \emptyset \wedge D_b = \emptyset \wedge D_p = \emptyset$
2. c is necessarily false at n if $D_b \neq \emptyset \wedge A_b = \emptyset \wedge A_p = \emptyset$.
3. c is possibly true at n if $A_b \cup A_p \neq \emptyset$.
4. c is possibly false at n if $D_b \cup D_p \neq \emptyset$.

In case (1), a contributor is one of the nodes in A_b. In case (3), one of the nodes in $A_b \cup A_p$ can be made a contributor by a linking process, based on that of UM NONLIN, that links in a contributor and links out potential clobberers. In case (4), CAPER may be able to establish c by the linking procedure. Otherwise, it will, as with case (3), have to retrieve a **patch** plan that achieves c, as described in the next section. If c is a filter condition, it will not be achieved.

6.2.7 Retrieving Patch and Substitute Plans

CAPER needs to retrieve a patch plan to achieve a goal condition for a missing or failing validation. CAPER needs to retrieve a a substitute plan for rereducing a goal due to a failing filter condition. Because, unlike PRIAR, CAPER lacks a generative planner, it must achieve a goal by going back to memory to find a (sub)plan from a source case in the KB. It does this as follows for a goal g:

1. A case feature is created for g.
2. The Plan Retriever is invoked with a new retrieval description containing g.
3. A source plan p_s is retrieved, mapped by the Plan Mapper, and instantiated as a new target plan p_t.
4. Plan p_t is merged into the overall target plan, as described in the following section.

Note that the plan p_t may serendipitously achieve other goals in the plan modification task queue. A potential enhancement to this process would be to combine features for multiple goals to be achieved into a single retrieval description. This could be used to explicitly search for and preferably find a patch/substitute plan that achieves as many of these goals as possible.

6.3 Plan Merger

This component takes a newly instantiated patch or substitute target plan and merges it into the overall target plan. CAPER has flexibility as to where merge a patch plan. Its actions that contribute some condition to make a validation hold must precede that validation's consumer. A substitute plan is initially inserted into the overall target plan in place of the plan it replaces. This is shown in Figure 6.3. When substituting target plan Plan-2 for target plan Plan-1 in the overall plan, the following must be checked, as is illustrated in the figure:

- Plan-2 should supply all e-conditions supplied by Plan-1.
- The overall plan should supply all e-preconditions of Plan-2.
- Plan-2 should not clobber any of the p-conditions of Plan-1.

The latter is checked by comparing the effects of actions in Plan-2 with the p-conditions of Plan-1. Those p-conditions that are clobbered are repaired as described in the previous section.

If any of the above requirements are not met, then additional refitting of the overall plan will be needed to reestablish failing or missing validations. It also may be possible to exploit existing contributors:

- contributors of e-preconditions of Plan-1 might be used as contributors for e-preconditions of Plan-2
- contributors of e-conditions of Plan-2 might be used as contributors for e-conditions of Plan-1

CAPER checks for these situations and modifies the validations accordingly. E-preconditions of Plan-1 and e-conditions of Plan-2 that are now unnecessary will be pruned later (they are added to list of plan modification tasks).

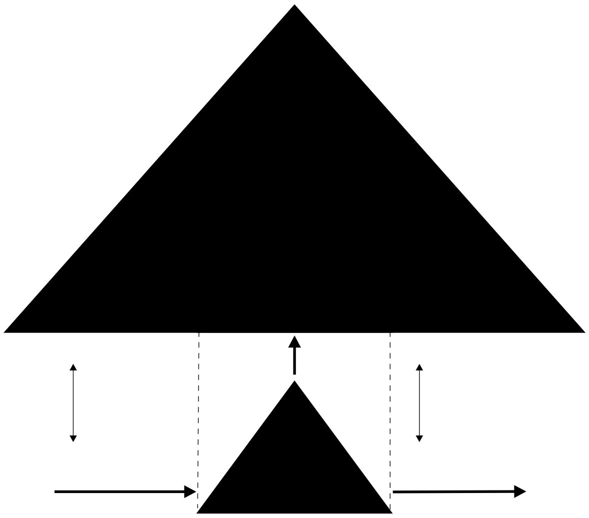

Figure 6.3: The checking of validations when merging a new subplan, Plan-2 a replacement for Plan-1 in the target plan.

6.4 Additional Plan Adaptation Issues

CAPER requires some causal domain knowledge for plan adaptation. For example, it needs to know the effect of an e-action to determine if that action clobbers some validation. Where possible, this causal domain knowledge is episodic in CAPER, rather than conceptual. For example, the (instantiated) effects of an e-action are known because the were stored by the planner. CAPER can access this information rather than looking up the generic effects of the action's type in some schema table. as UM NONLIN does. Only those effects that were "observed" in the e-action are available to CAPER. In addition, CAPER uses actual instantiated subplans rather than a task schema to "reduce" a goal.

6.5 Plan Storer

The Plan Storer module takes a complete target plan that has been debugged by the Plan Modifier and stores it in the KB. This process is fairly straightforward and will not be described in detail here. New PARKA frames and assertions are created for each e-plan (and subplan), e-action, plan validation, condition, goal, and plan variable. The new frames for each of these has a slot that points to its analogous source frame. For example, the frame for the root plan for Case-3 has *Plan-Pkg-Located-At-1.2* in its *source* slot. By saving this mapping between source and target frames, CAPER is essentially saving the analogy itself, which may prove useful in the future (though currently CAPER does not exploit this meta-knowledge).

The representations used are described in Section 3.4. The new case is also annotated with information about what version of CAPER created it, etc. The modified KB can be written to disk via PARKA's KB dumper mechanism.

6.6 Chapter Summary

This chapter described and presented an example of CAPER's Plan Checker, Plan Modifier, and Plan Storer modules.

Memory was accessed frequently during each of these phases: to check for out facts, to find patch/replacement plans, and to create target frames for a new plan/case.

Annotation verification and repair procedures based on those in PRIAR were adapted to the reuse-only framework of CAPER, in which a generative planner is not available. Plan validations are used to find applicability failures and create refit tasks. Instead of refit by generative techniques, CAPER goes back to memory to get a patch or replacement plan. This is made possible by CAPER's flexible retrieval mechanisms and case representations that allow pieces of plans to be accessed.

Chapter 7

Extended Examples

This section presents extended examples of CAPER's operation. Actual output from CAPER is shown, though parts of the output have been omitted for brevity in places where noted. The output has been annotated with brief comments prefaced by ";;". Note that symbols prefaced with a "#!" are PARKA frame names. Many of the data structures displayed are described in Appendix B.

These examples are taken from the UM TRANSLOG domain (Section A.1). Examples from the Car Construction and Protein Sequencing domains are presented in (Kettler *et al.*, 1994) and (Kettler and Darden, 1993), respectively.

7.1 Example 1: Delivering a Liquid Package within a City

This example, described in the text as Case-3, is shown here in greater detail (as space permits). The instantiated target plan is modified by having its subplans for loading and unloading a vehicle replaced by plans instantiated from subplans retrieved from the casebase. Although the goal is relatively simple, a lot of work is required to retrieve, map, instantiate, check and merge the three target plans.

```
;; CAPER is invoked with the target problem to solve (Case-3).
;; A liguid-type package, Pkg-1.3, is to be delivered from
;; City1-CL1 to City1-CL2.  (Pkg-1.3 was previously referred
;; to in the preceding text as ''Pkg-3.1'').
;; New frames are greated, along with the initial case feature
;; which is stored in a new problem description.

? (umt-test1)
THE CAPER CASE-BASED PLANNING SYSTEM
(Version 0.3.1.)
(c) 1995, Dept. of Computer Science
University of Maryland at College Park
```

```
UMT-PDESC Problem to be solved: LR-L-1112 (World: WORLD-RT1)

New problem is designated as case "Case-Make-Deliveries-3".
Defining new frames for case "Case-Make-Deliveries-3" objects...
Defining plan frame "Plan-Make-Deliveries-3"
Initializing world frames for situation #!Case-Make-Deliveries-3-InitSit
Defining initial features for case "Case-Make-Deliveries-3" objects...
Generated initial problem description (*target-pdesc*):
Target Problem TOP-PROB-1564 (LR-L-1112) (Class UMT-PDESC):
          Domain: :UM-TRANSLOG ("UM-Translog Domain")
            Case: #!Case-Make-Deliveries-3,  Plan: #!Plan-Make-Deliveries-3
          RPLANs: NIL
   Source PPLANS: NIL
   Target PPLANS: NIL
      Root PPLAN: NIL (#!Plan-Make-Deliveries-3)
     Subproblems: NIL
   Features: (UMT-GOAL-TRANSPORT-PKG-1575)
   Mappings: NIL
Feature UMT-GOAL-TRANSPORT-PKG-1575 (prob.fid: UMT-GOAL-TRANSPORT-PKG-1575):
            Class: UMT-GOAL-TRANSPORT-PKG
       Feature Of: :TARGET,  Derived From: NIL,  Derived Via: :PREDEFINED
           Facets: (FACET-1577 FACET-1578 FACET-1579 FACET-1581 FACET-1583
                    FACET-1585)
   Facet bindings: ((PLAN #!Plan-Make-Deliveries-3)
                    (GOAL #!Case-Make-Deliveries-3-Goal-1)
 (PKG #!Pkg-1.3) (ORGN #!City1-Cl1)
 (DEST #!City1-Cl2) (ISIT #!Case-Make-Deliveries-3-InitSit))

;; Next, an initial retrieval description and retrieval plan are created
;; to try and find a source plan for delivering a liquid-type package
;; from the source (City1-CL1) to the destination (City1-CL2).

New RPLAN RPLAN-1589 created for target problem.
<<< Retrieving plan using RPLAN RPLAN-1589...
Rtvl. Plan RPLAN-1589 (RPLAN)
    Derived-via: :AUTO
        Problem: TOP-PROB-1564
       Features: (UMT-GOAL-TRANSPORT-PKG-1575)
         RDESCs: (RDESC-1590)
         Status: NIL
    Succ. RDESC: NIL
   Sources Rtvd: NIL
   Srcs. Chosen: NIL
Rtvl. Description RDESC-1590
   Derived-from: NIL
    Derived-via: :AUTO
     Rtvl. Plan: RPLAN-1589
    Query Forms: NIL
    Query Vars.: NIL
   Query Source: NIL
   Query Status: NIL
```

```
   Query Result: NIL
   Features: (UMT-GOAL-TRANSPORT-PKG-1575)
   Mappings: NIL

;; CAPER begins searching the space of retrieval descriptions from this
;; initial retrieval description.  Search is set to be done best-first.

<<< Beginning RDESC search from initial rdescs: (RDESC-1590)
Current OPEN list:
((NODE-1591 RDESC-1590 NIL 0 NIL)) ;; shows node-id, rdesc-id, parent

;; The initial retrieval description is executed as a Parka Structure
;; Matcher probe graph, which fails.

<<< Iteration 1:
<<< Expanding New NODE-1591: Old RDESC-1590 (node NIL), New RDESC-1590
<<< Calling Parka Structure Matcher with query:
(:AND (#!instanceOf ?PKG-PVAR #!Plan-Variable) (#!var-Value= ?PKG-PVAR ?PKG)
 (#!thing= ?GOAL ?PKG-PVAR) (#!instanceOf ?DEST-PVAR #!Plan-Variable)
 (#!var-Value= ?DEST-PVAR #!City1-Cl2) (#!location= ?GOAL ?DEST-PVAR)
 (#!goal-of-Plan ?GOAL ?PLAN) (#!situation-Before= ?PLAN ?ISIT) (#!instanceOf
 ?SITVAL #!Situated-Value) (#!location-sit= ?PKG ?SITVAL)
 (#!sslot-Val= ?SITVAL
 #!City1-Cl1) (#!situation= ?SITVAL ?ISIT) (#!instanceOf ?PLAN #!E-Plan)
 (#!modality= ?GOAL #!True) (#!instanceOf ?GOAL #!CE-Located-At)
 (#!everyInstanceOf ?PKG #!Liquid-Package) (#!instanceOf ?ISIT
 #!Initial-Situation))
<<< PSM query 1 result is (0 bindings):
NIL
<<< New query forms were: NIL

;; Retrieval description operators are applied to modify the failing RDESC.
3 Successor nodes generated for node NODE-1591.

<<< Generating New NODE-1595: Old RDESC-1590 (node NODE-1591), New NIL
      Via New
      RDOP-1592 (RDOP-GENERALIZE-FACET) [Status NIL, Score 0, Cost 1] Replace
      Type FVT-1588 ((#!everyInstanceOf #!Liquid-Package (#!Liquid-Package
      #!Package))) of FACET-1579 with: ((#!everyInstanceOf #!Package
      (#!Liquid-Package #!Package)))

<<< Generating New NODE-1596: Old RDESC-1590 (node NODE-1591), New NIL
      Via New
      RDOP-1593 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 1, Cost 1]
      Variablize FACET-1581 (UMT-FCT-LOC-GEO) of UMT-GOAL-TRANSPORT-PKG-1575
      Facet Var Types: ((#!everyInstanceOf NIL (#!City-Location)))

<<< Generating New NODE-1597: Old RDESC-1590 (node NODE-1591), New NIL
      Via New RDOP-1594 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 1,
      Cost 1]
    Variablize FACET-1583 (UMT-FCT-LOC-GEO) of UMT-GOAL-TRANSPORT-PKG-1575
```

```
    Facet Var Types: ((#!everyInstanceOf NIL (#!City-Location)))

Current OPEN list: ((NODE-1595 RDESC-1590 RDOP-1592 1 0)
   (NODE-1597 RDESC-1590 RDOP-1594 1 1) (NODE-1596 RDESC-1590 RDOP-1593 1 1))

<<< Iteration 2:

;; One of the new retrieval descriptions is chosen to try.  It has the
;; Package facet value generalized from Liquid-Package to Package.
;; This retrieval description succeeds with 2 sets of bindings.

<<< Expanding New NODE-1595: Old RDESC-1590 (node NODE-1591), New NIL Via New
      RDOP-1592 (RDOP-GENERALIZE-FACET) [Status 2, Score 0, Cost 1]
      Replace Type
      FVT-1588 ((#!everyInstanceOf #!Liquid-Package (#!Liquid-Package
      #!Package))) of FACET-1579 with: ((#!everyInstanceOf #!Package
      (#!Liquid-Package #!Package)))

<<< Calling Parka Structure Matcher with query: (:AND (#!instanceOf ?PKG-PVAR
#!Plan-Variable) (#!var-Value= ?PKG-PVAR ?PKG) (#!thing= ?GOAL ?PKG-PVAR)
(#!instanceOf ?DEST-PVAR #!Plan-Variable)
(#!var-Value= ?DEST-PVAR #!City1-Cl2)
(#!location= ?GOAL ?DEST-PVAR) (#!goal-of-Plan ?GOAL ?PLAN)
(#!situation-Before= ?PLAN ?ISIT) (#!instanceOf ?SITVAL #!Situated-Value)
(#!location-sit= ?PKG ?SITVAL) (#!sslot-Val= ?SITVAL #!City1-Cl1)
(#!situation= ?SITVAL ?ISIT)
(#!instanceOf ?ISIT #!Initial-Situation) (#!everyInstanceOf ?PKG #!Package)
(#!instanceOf ?GOAL #!CE-Located-At) (#!modality= ?GOAL #!True) (#!instanceOf
?PLAN #!E-Plan))

<<< PSM query 2 result is (2 bindings): (((?GOAL
. #!Plan-CE-Located-At-1.2-Goal-1) (?PLAN . #!Plan-CE-Located-At-1.2)
(?PKG-PVAR
. #!PVAR-P1) (?SITVAL . #!SitVal-1333) (?ISIT
. #!Case-Make-Deliveries-1-InitSit) (?PKG . #!Pkg-1.1) (?DEST-PVAR
. #!PVAR-L12B)) ((?GOAL . #!Plan-CE-Located-At-1.3-Goal-1) (?PLAN
. #!Plan-CE-Located-At-1.3) (?PKG-PVAR . #!PVAR-P1) (?SITVAL . #!SitVal-1333)
(?ISIT . #!Case-Make-Deliveries-1-InitSit) (?PKG . #!Pkg-1.1) (?DEST-PVAR
. #!PVAR-L12B)))
<<< New query forms were: ((#!everyInstanceOf ?PKG #!Package))

<<< Done with RDESC Search.  2 nodes expanded
    Search Status is SUCCESS.

;; The similarity heuristic ``choose subplan over its parent plan'' is
;; applied to select a source plan.

<<< Retrieved 2 candidate source plans: (#!Plan-CE-Located-At-1.2
   #!Plan-CE-Located-At-1.3) Ignoring Plan #!Plan-CE-Located-At-1.3,
   a superplan of #!Plan-CE-Located-At-1.2.
```

```
<<< Selecting high scoring plan "Plan-CE-Located-At-1.2" (SDESC SPLAN-1671),
         score = NIL: Features: (UMT-GOAL-TRANSPORT-PKG-1628) Mappings:
         ((UMT-GOAL-TRANSPORT-PKG-1575 . UMT-GOAL-TRANSPORT-PKG-1628))

Actions of source plan #!Plan-CE-Located-At-1.2: (#!Do-Nothing-1.1
#!Do-Nothing-1.2 #!P-Collect-Fees-1.1 #!Do-Nothing-1.3
#!P-Open-Door-1.1 #!P-Load-Package-1.1 #!P-Close-Door-1.1
#!P-Move-Vehicle-1.1 #!P-Open-Door-1.2 #!P-Unload-Package-1.1
#!P-Close-Door-1.2 #!Do-Nothing-1.4 #!P-Deliver-1.1)

PARKA (PACK) Current Query Counts (Details Collected=T):
 Total Queries: 100
 1 Var Queries: 19
 2 Var Queries: 2
 Other Queries: 79
 Forms / Query: 1.32
 Vars  / Query: 0.33
 Call-SM calls: 2
<<< Mapping source plan #!Plan-CE-Located-At-1.2 to problem TOP-PROB-1564...

;; The Plan Mapper is invoked to map unbound variables.

Ignoring variable #!PVAR-RP with target value #!Liquid-Package.
Ignoring variable #!PVAR-L12B with target value #!City1-Cl2.
Ignoring variable #!PVAR-P1 with target value #!Pkg-1.3.
Getting initial domains for unbound variables...

;; Unary constraints are checked via the Parka Structure Matcher.
;; The variable domains are pruned.

Checking hard unary vconstraint for #!PVAR-C1B: #S(VCON :ID VCON-1.4)

<<< Calling Parka Structure Matcher with query:
(:AND (#!everyInstanceOf ?QVAR-1672 #!City)
      (#!in-City #!City1-Cl2 ?QVAR-1672))
<<< PSM query 3 result is (1 bindings):
(((?QVAR-1672 . #!City1)))
QUERY-VCON Result: (((#!PVAR-C1B . #!City1)))
Old domain values of #!PVAR-C1B: ((#!City3 0) (#!City2 0) (#!City1 0))
Supported values of #!PVAR-C1B: (#!City1)
New domain values of #!PVAR-C1B: ((#!City1 1))

Checking hard unary vconstraint for #!PVAR-C1B: #S(VCON :ID VCON-1.8)

<<< Calling Parka Structure Matcher with query:
(:AND (#!everyInstanceOf ?QVAR-1690 #!City)
      (#!in-City #!City1-Cl1 ?QVAR-1690))
<<< PSM query 4 result is (1 bindings):
(((?QVAR-1690 . #!City1)))
QUERY-VCON Result: (((#!PVAR-C1B . #!City1)))
Old domain values of #!PVAR-C1B: ((#!City1 1))
```

```
Supported values of #!PVAR-C1B: (#!City1)
New domain values of #!PVAR-C1B: ((#!City1 2))
Ignoring variable #!PVAR-C1B with singleton domain value (#!City1 2).

Checking hard unary vconstraint for #!PVAR-C1A: #S(VCON :ID VCON-1.27)

<<< Calling Parka Structure Matcher with query:
(:AND (#!everyInstanceOf ?QVAR-1688 #!City)
      (#!in-City #!City1-Cl2 ?QVAR-1688))
<<< PSM query 5 result is (1 bindings):
(((?QVAR-1688 . #!City1)))
QUERY-VCON Result: (((#!PVAR-C1A . #!City1)))
Old domain values of #!PVAR-C1A: ((#!City3 0) (#!City2 0) (#!City1 0))
Supported values of #!PVAR-C1A: (#!City1)
New domain values of #!PVAR-C1A: ((#!City1 1))
Ignoring variable #!PVAR-C1A with singleton domain value (#!City1 1).

Checking hard unary vconstraint for #!PVAR-T1: #S(VCON :ID VCON-1.10)

<<< Calling Parka Structure Matcher with query:
(:AND (#!everyInstanceOf ?QVAR-1673 #!Vehicle) (#!instanceOf ?SITVAL
#!Situated-Value) (#!available-sit?  ?QVAR-1673 ?SITVAL)
(#!sslot-Val= ?SITVAL
#!True) (#!situation= ?SITVAL #!Case-Make-Deliveries-3-InitSit))
<<< PSM query 6 result is (2 bindings): (((?QVAR-1673 . #!Truck-1) (?SITVAL
. #!SitVal-1568)) ((?QVAR-1673 . #!Truck-3) (?SITVAL . #!SitVal-1569)))
QUERY-VCON Result: (((#!PVAR-T1 . #!Truck-3)) ((#!PVAR-T1 . #!Truck-1)))
Old domain values of #!PVAR-T1: ((#!Truck-1 0) (#!Truck-4 0) (#!Truck-3 0))
Supported values of #!PVAR-T1: (#!Truck-3 #!Truck-1)
New domain values of #!PVAR-T1: ((#!Truck-3 1) (#!Truck-1 1))

Checking soft unary vconstraint for #!PVAR-T1: #S(VCON :ID VCON-1.18)

<<< Calling Parka Structure Matcher with query:
(:AND (#!everyInstanceOf ?QVAR-1689 #!Vehicle) (#!instanceOf ?SITVAL
 #!Situated-Value) (#!location-sit= ?QVAR-1689 ?SITVAL) (#!sslot-Val= ?SITVAL
 #!City1-Cl1) (#!situation= ?SITVAL #!Case-Make-Deliveries-3-InitSit))

<<< PSM query 7 result is (2 bindings): (((?QVAR-1689 . #!Truck-1) (?SITVAL
 . #!SitVal-1565)) ((?QVAR-1689 . #!Truck-3) (?SITVAL . #!SitVal-1566)))
QUERY-VCON Result: (((#!PVAR-T1 . #!Truck-3)) ((#!PVAR-T1 . #!Truck-1)))
Old domain values of #!PVAR-T1: ((#!Truck-3 1) (#!Truck-1 1))
Supported values of #!PVAR-T1: (#!Truck-3 #!Truck-1)
New domain values of #!PVAR-T1: ((#!Truck-3 2) (#!Truck-1 2))

CHECK-UNARY: #!PVAR-L11C domain:
Checking hard unary vconstraint for #!PVAR-L11C: #S(VCON :ID VCON-1.9)

<<< Calling Parka Structure Matcher with query:
(:AND (#!everyInstanceOf ?QVAR-1700 #!City-Location) (#!in-City ?QVAR-1700
#!City1))
```

```
<<< PSM query 8 result is (6 bindings): (((?QVAR-1700 . #!City1-Cl2))
((?QVAR-1700 . #!City1-Cl1)) ((?QVAR-1700 . #!City1-Ap2)) ((?QVAR-1700
. #!City1-Ap1)) ((?QVAR-1700 . #!City1-Ts2)) ((?QVAR-1700 . #!City1-Ts1)))

QUERY-VCON Result: (((#!PVAR-L11C . #!City1-Ts1)) ((#!PVAR-L11C
. #!City1-Ts2)) ((#!PVAR-L11C . #!City1-Ap1)) ((#!PVAR-L11C
. #!City1-Ap2)) ((#!PVAR-L11C . #!City1-Cl1)) ((#!PVAR-L11C
. #!City1-Cl2)))

Old domain values of #!PVAR-L11C: ((#!City3-Cl1 0) (#!City2-Cl1 0)
(#!City1-Cl2 0) (#!City1-Cl1 0) (#!Region1-Ap1 0)(#!City3-Ap1 0)
(#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0) (#!Region1-Ts1 0)
(#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0) (#!City1-Ts1 0))

Supported values of #!PVAR-L11C: (#!City1-Ts1 #!City1-Ts2 #!City1-Ap1
#!City1-Ap2 #!City1-Cl1 #!City1-Cl2)
New domain values of #!PVAR-L11C: ((#!City1-Ts1 1) (#!City1-Ts2 1)
(#!City1-Ap1 1) (#!City1-Ap2 1) (#!City1-Cl1 1) (#!City1-Cl2 1))

CHECK-UNARY: #!PVAR-RT domain:
Checking hard unary vconstraint for #!PVAR-RT: #S(VCON :ID VCON-1.11)

<<< Calling Parka Structure Matcher with query: (:AND (#!ancestor
?QVAR-1679 #!Vehicle) (#!pv-Compatible= #!Liquid-Package ?QVAR-1679))
<<< PSM query 9 result is (8 bindings): (((?QVAR-1679
. #!Armored-Refrigerated-Tanker-Traincar)) ((?QVAR-1679
. #!Armored-Tanker-Traincar)) ((?QVAR-1679
. #!Refrigerated-Tanker-Traincar)) ((?QVAR-1679 . #!Tanker-Traincar))
((?QVAR-1679 . #!Armored-Refrigerated-Tanker-Truck)) ((?QVAR-1679
. #!Armored-Tanker-Truck)) ((?QVAR-1679
. #!Refrigerated-Tanker-Truck)) ((?QVAR-1679 . #!Tanker-Truck)))
QUERY-VCON Result: (((#!PVAR-RT . #!Tanker-Truck)) ((#!PVAR-RT
. #!Refrigerated-Tanker-Truck)) ((#!PVAR-RT . #!Armored-Tanker-Truck))
;; etc.))

Supported values of #!PVAR-RT: (#!Tanker-Truck
 #!Refrigerated-Tanker-Truck #!Armored-Tanker-Truck
 #!Armored-Refrigerated-Tanker-Truck #!Tanker-Traincar
 #!Refrigerated-Tanker-Traincar #!Armored-Tanker-Traincar
 #!Armored-Refrigerated-Tanker-Traincar)

New domain values of #!PVAR-RT:
((#!Armored-Refrigerated-Tanker-Traincar 1)
(#!Armored-Refrigerated-Tanker-Truck 1) (#!Armored-Tanker-Traincar 1)
(#!Armored-Tanker-Truck 1) (#!Refrigerated-Tanker-Traincar 1)
(#!Refrigerated-Tanker-Truck 1) (#!Tanker-Traincar 1) (#!Tanker-Truck
1)) PVAR Scores: ((#!PVAR-RT . 1) (#!PVAR-T1 . 6)) PVAR Scores:
((#!PVAR-L11C . 2) (#!PVAR-L11A . 2))

Variables to bind (in order): (#!PVAR-T1 #!PVAR-RT #!PVAR-L11C #!PVAR-L11A)
#!PVAR-T1 - sval: #!Truck-1, tval: NIL, tdom: ((#!Truck-3 2) (#!Truck-1 2))
```

```
#!PVAR-RT - sval: #!Regular-Truck, tval: NIL, tdom:
((#!Armored-Refrigerated-Tanker-Traincar 1)
 (#!Armored-Refrigerated-Tanker-Truck 1) (#!Armored-Tanker-Traincar 1)
 (#!Armored-Tanker-Truck 1)
(#!Refrigerated-Tanker-Traincar 1) (#!Refrigerated-Tanker-Truck 1)
(#!Tanker-Traincar 1) (#!Tanker-Truck 1))

#!PVAR-L11C - sval: #!City1-Cl1, tval: NIL, tdom: ((#!City1-Ts1 1)
(#!City1-Ts2 1) (#!City1-Ap1 1) (#!City1-Ap2 1) (#!City1-Cl1 1)
(#!City1-Cl2 1))

#!PVAR-L11A - sval: #!City1-Cl1, tval: NIL, tdom: ((#!City3-Cl1 0)
(#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 0) (#!Region1-Ap1 0)
(#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0)
(#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0)
(#!City1-Ts1 0))

;; Done with pruning domains via unary constraints.  Now plan variables will
;; be bound by satisfying binary plan variable constraints.

Active Constraints:
(#S(VCON :ID VCON-1.1 :STATUS NIL :SOLVE-P T :VDTN #!Plan-Vdtn-1.1 :VARS
(#!PVAR-RT #!PVAR-T1) :CFORM (Q-HAS-TYPE #!PVAR-T1 #!PVAR-RT) :CMODALITY T
:QFORM (Q-HAS-TYPE ?QVAR-1680 ?QVAR-1681) :QSUBS ((?QVAR-1681 . #!PVAR-RT)
(?QVAR-1680 . #!PVAR-T1)) :SIT :INITIAL)

#S(VCON :ID VCON-1.3 :STATUS NIL :SOLVE-P T :VDTN #!Plan-Vdtn-1.3
         :VARS (#!PVAR-L11C #!PVAR-T1) :CFORM (Q-LOCATED-AT #!PVAR-T1
         #!PVAR-L11C #!Case-Make-Deliveries-3-InitSit) :CMODALITY T
         :QFORM (Q-LOCATED-AT ?QVAR-1682 ?QVAR-1683
         #!Case-Make-Deliveries-3-InitSit) :QSUBS ((?QVAR-1683
         . #!PVAR-L11C) (?QVAR-1682 . #!PVAR-T1)) :SIT :INITIAL)

#S(VCON :ID VCON-1.2 :STATUS NIL :SOLVE-P T :VDTN #!Plan-Vdtn-1.2
         :VARS (#!PVAR-L11A #!PVAR-T1) :CFORM (Q-LOCATED-AT #!PVAR-T1
         #!PVAR-L11A #!Case-Make-Deliveries-3-InitSit) :CMODALITY T
         :QFORM (Q-LOCATED-AT ?QVAR-1694 ?QVAR-1695
         #!Case-Make-Deliveries-3-InitSit) :QSUBS ((?QVAR-1695
         . #!PVAR-L11A) (?QVAR-1694 . #!PVAR-T1)) :SIT :INITIAL))

;; The Forward-Checking constraint satisfaction search procedure is invoked.

Initializing for new FCS...

Beginning FCS using d= ((#!PVAR-T1 (#!Truck-3 2) (#!Truck-1 2))
(#!PVAR-RT (#!Armored-Refrigerated-Tanker-Traincar 1)
(#!Armored-Refrigerated-Tanker-Truck 1) (#!Armored-Tanker-Traincar 1)
(#!Armored-Tanker-Truck 1) (#!Refrigerated-Tanker-Traincar 1)
(#!Refrigerated-Tanker-Truck 1) (#!Tanker-Traincar 1) (#!Tanker-Truck
1)) (#!PVAR-L11C (#!City1-Ts1 1) (#!City1-Ts2 1) (#!City1-Ap1 1)
```

```
(#!City1-Ap2 1) (#!City1-Cl1 1) (#!City1-Cl2 1)) (#!PVAR-L11A
(#!City3-Cl1 0) (#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 0)
(#!Region1-Ap1 0) (#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0)
(#!City1-Ap1 0) (#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1 0)
(#!City1-Ts2 0) (#!City1-Ts1 0)))

FCS choosing NIL = NIL by selecting node #S(FCSN :ID :DUMMY :K 0 :ZK
NIL) :STATUS :UNEXPANDED) from FCS Open List.

CHECK-FORWARD: k=1, var=#!PVAR-T1,

FCS expanding node #S(FCSN :ID :DUMMY :K 0 :ZK NIL)
                           :STATUS :EXPANDED) (count 1)
FCS adding new node to open: #S(FCSN :ID NODE-1701 :K 1 :ZK (#!Truck-3 2))
                           :STATUS :UNEXPANDED) (count 2)
FCS adding new node to open: #S(FCSN :ID NODE-1702 :K 1 :ZK (#!Truck-1 2))
                           :STATUS :UNEXPANDED) (count 3)

FCS choosing #!PVAR-T1 = (#!Truck-3 2) by selecting node #S(FCSN :ID
NODE-1701 :K 1 :ZK (#!Truck-3 2) :STATUS :UNEXPANDED) from FCS Open
List.

;; Invoking procedure check-forward to check compatibility with
PVAR-T1 = Truck-3

CHECK-FORWARD: k=2, var=#!PVAR-RT

REVISE: i=2 (#!PVAR-RT), j=1 (#!PVAR-T1) dom(i)=
(#!Armored-Refrigerated-Tanker-Traincar 1)
(#!Armored-Refrigerated-Tanker-Truck 1) (#!Armored-Tanker-Traincar 1)
(#!Armored-Tanker-Truck 1) (#!Refrigerated-Tanker-Traincar 1)
(#!Refrigerated-Tanker-Truck 1) (#!Tanker-Traincar 1) (#!Tanker-Truck
1)), dom(j)=((#!Truck-3 2))

;; Check binary constraints involving these two plan variables
CHECK-BINARY: #!PVAR-RT and #!PVAR-T1

Checking hard binary vconstraint for #!PVAR-RT, #!PVAR-T1: #S(VCON :ID
VCON-1.1)

<<< Calling Parka Structure Matcher with query:

(:AND (#!ancestor ?QVAR-1681 #!Vehicle) (#!everyInstanceOf #!Truck-3
?QVAR-1681))

<<< PSM query 10 result is (5 bindings): (((?QVAR-1681
. #!Tanker-Vehicle)) ((?QVAR-1681 . #!Tanker-Truck)) ((?QVAR-1681
. #!Truck)) ((?QVAR-1681 . #!Land-Vehicle)) ((?QVAR-1681
. #!Vehicle))) QUERY-VCON Result: (((#!PVAR-RT . #!Vehicle) (#!PVAR-T1
. #!Truck-3)) ((#!PVAR-RT . #!Land-Vehicle) (#!PVAR-T1 . #!Truck-3))
((#!PVAR-RT . #!Truck) (#!PVAR-T1 . #!Truck-3)) ((#!PVAR-RT
```

```
. #!Tanker-Truck) (#!PVAR-T1 . #!Truck-3)) ((#!PVAR-RT
. #!Tanker-Vehicle) (#!PVAR-T1 . #!Truck-3)))

Old domain values of #!PVAR-RT:
((#!Armored-Refrigerated-Tanker-Traincar 1)
(#!Armored-Refrigerated-Tanker-Truck 1) (#!Armored-Tanker-Traincar 1)
(#!Armored-Tanker-Truck 1) (#!Refrigerated-Tanker-Traincar 1)
(#!Refrigerated-Tanker-Truck 1) (#!Tanker-Traincar 1) (#!Tanker-Truck
1)) Supported values of #!PVAR-RT and #!PVAR-T1: ((#!Tanker-Truck
. #!Truck-3)) New domain values of #!PVAR-RT: ((#!Tanker-Truck 2))

REVISE: i=3 (#!PVAR-L11C), j=1 (#!PVAR-T1) dom(i)= ((#!City1-Ts1 1)
(#!City1-Ts2 1) (#!City1-Ap1 1) (#!City1-Ap2 1) (#!City1-Cl1 1)
(#!City1-Cl2 1)), dom(j)=((#!Truck-3 2))

CHECK-BINARY: #!PVAR-L11C and #!PVAR-T1

Checking soft binary vconstraint for #!PVAR-L11C, #!PVAR-T1: #S(VCON
:ID VCON-1.3)

<<< Calling Parka Structure Matcher with query: (:AND
(#!everyInstanceOf ?QVAR-1683 #!City-Location) (#!instanceOf ?SITVAL
#!Situated-Value) (#!location-sit= #!Truck-3 ?SITVAL) (#!sslot-Val=
?SITVAL ?QVAR-1683) (#!situation= ?SITVAL
#!Case-Make-Deliveries-3-InitSit))
<<< PSM query 11 result is (1 bindings):
(((?SITVAL . #!SitVal-1566) (?QVAR-1683 . #!City1-Cl1)))
QUERY-VCON Result: (((#!PVAR-L11C . #!City1-Cl1) (#!PVAR-T1
. #!Truck-3)))
Old domain values of #!PVAR-L11C: ((#!City1-Ts1 1) (#!City1-Ts2 1)
(#!City1-Ap1 1) (#!City1-Ap2 1) (#!City1-Cl1 1) (#!City1-Cl2 1))
Supported values of #!PVAR-L11C and #!PVAR-T1: ((#!City1-Cl1
. #!Truck-3)) New domain values of #!PVAR-L11C: ((#!City1-Ts1 1)
(#!City1-Ts2 1) (#!City1-Ap1 1) (#!City1-Ap2 1) (#!City1-Cl1 2)
(#!City1-Cl2 1))

REVISE: i=4 (#!PVAR-L11A), j=1 (#!PVAR-T1) dom(i)= ((#!City3-Cl1 0)
(#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 0) (#!Region1-Ap1 0)
(#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0)
(#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0)
(#!City1-Ts1 0)), dom(j)=((#!Truck-3 2))

CHECK-BINARY: #!PVAR-L11A and #!PVAR-T1

Checking soft binary vconstraint for #!PVAR-L11A, #!PVAR-T1: #S(VCON
:ID VCON-1.2)

<<< Calling Parka Structure Matcher with query: (:AND
(#!everyInstanceOf ?QVAR-1695 #!City-Location) (#!instanceOf ?SITVAL
#!Situated-Value) (#!location-sit= #!Truck-3 ?SITVAL) (#!sslot-Val=
?SITVAL ?QVAR-1695) (#!situation= ?SITVAL
```

```
#!Case-Make-Deliveries-3-InitSit)) <<< PSM query 12 result is (1
bindings): (((?SITVAL . #!SitVal-1566) (?QVAR-1695 . #!City1-Cl1)))
QUERY-VCON Result: (((#!PVAR-L11A . #!City1-Cl1) (#!PVAR-T1
. #!Truck-3)))
Old domain values of #!PVAR-L11A: ((#!City3-Cl1 0) (#!City2-Cl1 0)
(#!City1-Cl2 0) (#!City1-Cl1 0) (#!Region1-Ap1 0) (#!City3-Ap1 0)
(#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0) (#!Region1-Ts1 0)
(#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0) (#!City1-Ts1 0))
Supported values of #!PVAR-L11A and #!PVAR-T1: ((#!City1-Cl1
. #!Truck-3))
New domain values of #!PVAR-L11A: ((#!City3-Cl1 0) (#!City2-Cl1 0)
(#!City1-Cl2 0) (#!City1-Cl1 1) (#!Region1-Ap1 0) (#!City3-Ap1 0)
(#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0) (#!Region1-Ts1 0)
(#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0) (#!City1-Ts1 0))

FCS expanding node #S(FCSN :ID NODE-1701 :K 1 :ZK (#!Truck-3 2)
                           :STATUS :EXPANDED) (count 2) FCS adding new
                           node to open: #S(FCSN :ID NODE-1703 :K 2
                           :ZK (#!Tanker-Truck 2) :STATUS :UNEXPANDED)
                           (count 3))

FCS choosing #!PVAR-T1 = (#!Truck-1 2) by selecting node #S(FCSN :ID
                          NODE-1702 :K 1 :ZK (#!Truck-1 2) :STATUS
                          :UNEXPANDED) from FCS Open List.

CHECK-FORWARD: k=2, var=#!PVAR-RT

REVISE: i=2 (#!PVAR-RT), j=1 (#!PVAR-T1) dom(i)=
        ((#!Armored-Refrigerated-Tanker-Traincar 1)
        (#!Armored-Refrigerated-Tanker-Truck 1)
        (#!Armored-Tanker-Traincar 1) (#!Armored-Tanker-Truck 1)
        (#!Refrigerated-Tanker-Traincar 1)
        (#!Refrigerated-Tanker-Truck 1) (#!Tanker-Traincar 1)
        (#!Tanker-Truck 1)), dom(j)=((#!Truck-1 2))

CHECK-BINARY: #!PVAR-RT and #!PVAR-T1

Checking hard binary vconstraint for #!PVAR-RT, #!PVAR-T1: #S(VCON :ID
VCON-1.1 :STATUS)

<<< Calling Parka Structure Matcher with query:
(:AND (#!ancestor ?QVAR-1681 #!Vehicle) (#!everyInstanceOf #!Truck-1
?QVAR-1681))
<<< PSM query 13 result is (5 bindings):
(((?QVAR-1681 . #!Regular-Vehicle)) ((?QVAR-1681 . #!Regular-Truck))
 ((?QVAR-1681 . #!Truck)) ((?QVAR-1681 . #!Land-Vehicle)) ((?QVAR-1681
 . #!Vehicle))) QUERY-VCON Result: (((#!PVAR-RT . #!Vehicle)
 (#!PVAR-T1 . #!Truck-1)) ((#!PVAR-RT . #!Land-Vehicle) (#!PVAR-T1
 . #!Truck-1)) ((#!PVAR-RT . #!Truck) (#!PVAR-T1 . #!Truck-1))
 ((#!PVAR-RT . #!Regular-Truck) (#!PVAR-T1 . #!Truck-1)) ((#!PVAR-RT
 . #!Regular-Vehicle) (#!PVAR-T1 . #!Truck-1)))
```

```
Old domain values of #!PVAR-RT:
((#!Armored-Refrigerated-Tanker-Traincar 1)
(#!Armored-Refrigerated-Tanker-Truck 1) (#!Armored-Tanker-Traincar 1)
(#!Armored-Tanker-Truck 1) (#!Refrigerated-Tanker-Traincar 1)
(#!Refrigerated-Tanker-Truck 1) (#!Tanker-Traincar 1) (#!Tanker-Truck
1))

Supported values of #!PVAR-RT and #!PVAR-T1: NIL
New domain values of #!PVAR-RT: NIL

FCS node #S(FCSN :ID NODE-1702 :K 1 :ZK (#!Truck-1 2)
                 :STATUS :FAILURE) failed check-forward.

;; The choice of PVAR-T1 = Truck-1 fails because no value for PVAR-RT
;; is compatible.

FCS choosing #!PVAR-RT = (#!Tanker-Truck 2) by selecting node #S(FCSN :ID
                          NODE-1703 :K 2 :ZK (#!Tanker-Truck 2) :STATUS
                          :UNEXPANDED) from FCS Open List.

CHECK-FORWARD: k=3, var=#!PVAR-L11C

REVISE: i=3 (#!PVAR-L11C), j=2 (#!PVAR-RT) dom(i)=((#!City1-Ts1 1)
        (#!City1-Ts2 1) (#!City1-Ap1 1) (#!City1-Ap2 1) (#!City1-Cl1
        2) (#!City1-Cl2 1)), dom(j)=((#!Tanker-Truck 2))

CHECK-BINARY: #!PVAR-L11C and #!PVAR-RT

REVISE: i=4 (#!PVAR-L11A), j=2 (#!PVAR-RT) dom(i)=((#!City3-Cl1 0)
        (#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 1) (#!Region1-Ap1
        0) (#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0)
        (#!City1-Ap1 0) (#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1
        0) (#!City1-Ts2 0) (#!City1-Ts1 0)), dom(j)=((#!Tanker-Truck
        2))

CHECK-BINARY: #!PVAR-L11A and #!PVAR-RT
FCS expanding node #S(FCSN :ID NODE-1703 :K 2 :ZK (#!Tanker-Truck 2)
                           :STATUS :EXPANDED) (count 3)
FCS adding new node to open: #S(FCSN :ID NODE-1704 :K 3 :ZK (#!City1-Cl1 2)
                                     :STATUS :UNEXPANDED) (count 4)

;; ETC. (more nodes added to open)

FCS adding new node to open: #S(FCSN :ID NODE-1709 :K 3 :ZK (#!City1-Cl2 1)
                                     :STATUS :UNEXPANDED) (count 9)

FCS choosing #!PVAR-L11C = (#!City1-Cl1 2) by selecting node #S(FCSN :ID
                            NODE-1704 :K 3 :ZK (#!City1-Cl1 2) :STATUS
                            :UNEXPANDED) from FCS Open List.
```

```
CHECK-FORWARD: k=4, var=#!PVAR-L11A

REVISE: i=4 (#!PVAR-L11A), j=3 (#!PVAR-L11C) dom(i)=((#!City3-Cl1 0)
        (#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 1) (#!Region1-Ap1 0)
        (#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0)
        (#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0)
        (#!City1-Ts1 0)), dom(j)=((#!City1-Cl1 2))

CHECK-BINARY: #!PVAR-L11A and #!PVAR-L11C

FCS expanding node #S(FCSN :ID NODE-1704 :K 3 :ZK (#!City1-Cl1 2)
                           :STATUS :EXPANDED) (count 4)
FCS adding new node to open: #S(FCSN :ID NODE-1710 :K 4 :ZK (#!City1-Cl1 1)
                                     :STATUS :UNEXPANDED) (count 10)

;; ETC. (more nodes added to open)

FCS adding new node to open: #S(FCSN :ID NODE-1723 :K 4 :ZK (#!City1-Ts1 0)
                                     :STATUS :UNEXPANDED) (count 23)

FCS choosing #!PVAR-L11C = (#!City1-Ts1 1) by selecting node #S(FCSN :ID
                            NODE-1705 :K 3 :ZK (#!City1-Ts1 1) :STATUS
                            :UNEXPANDED) from FCS Open List.

CHECK-FORWARD: k=4, var=#!PVAR-L11A

REVISE: i=4 (#!PVAR-L11A), j=3 (#!PVAR-L11C) dom(i)=((#!City3-Cl1 0)
        (#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 1) (#!Region1-Ap1 0)
        (#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0)
        (#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0)
        (#!City1-Ts1 0)), dom(j)=((#!City1-Ts1 1))

CHECK-BINARY: #!PVAR-L11A and #!PVAR-L11C

FCS expanding node #S(FCSN :ID NODE-1705 :K 3 :ZK (#!City1-Ts1 1)
                           :STATUS :EXPANDED) (count 5)
FCS adding new node to open: #S(FCSN :ID NODE-1724 :K 4 :ZK (#!City1-Cl1 1)
                                     :STATUS :UNEXPANDED) (count 24)

;; ETC. (more nodes added to open)

FCS adding new node to open: #S(FCSN :ID NODE-1737 :K 4 :ZK (#!City1-Ts1 0)
                                     :STATUS :UNEXPANDED) (count 37)

FCS choosing #!PVAR-L11C = (#!City1-Ts2 1) by selecting node #S(FCSN :ID
                            NODE-1706 :K 3 :ZK (#!City1-Ts2 1)


CHECK-FORWARD: k=4, var=#!PVAR-L11A

REVISE: i=4 (#!PVAR-L11A), j=3 (#!PVAR-L11C) dom(i)=((#!City3-Cl1 0)
```

```
        (#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 1) (#!Region1-Ap1 0)
        (#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0)
        (#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0)
        (#!City1-Ts1 0)), dom(j)=((#!City1-Ts2 1))

CHECK-BINARY: #!PVAR-L11A and #!PVAR-L11C

FCS expanding node #S(FCSN :ID NODE-1706 :K 3 :ZK (#!City1-Ts2 1)
                           :STATUS :EXPANDED) (count 6)
FCS adding new node to open: #S(FCSN :ID NODE-1738 :K 4 :ZK (#!City1-Cl1 1)
                                     :STATUS :UNEXPANDED) (count 38)
;; ETC. (more nodes added to open)

FCS adding new node to open: #S(FCSN :ID NODE-1751 :K 4 :ZK (#!City1-Ts1 0)
                                     :STATUS :UNEXPANDED) (count 51)

FCS choosing #!PVAR-L11C = (#!City1-Ap1 1) by selecting node #S(FCSN :ID
                            NODE-1707 :K 3 :ZK (#!City1-Ap1 1) :STATUS
                            :UNEXPANDED) from FCS Open List.

CHECK-FORWARD: k=4, var=#!PVAR-L11A

REVISE: i=4 (#!PVAR-L11A), j=3 (#!PVAR-L11C) dom(i)=((#!City3-Cl1 0)
        (#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 1) (#!Region1-Ap1 0)
        (#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0)
        (#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0)
        (#!City1-Ts1 0)), dom(j)=((#!City1-Ap1 1))

CHECK-BINARY: #!PVAR-L11A and #!PVAR-L11C
FCS expanding node #S(FCSN :ID NODE-1707 :K 3 :ZK (#!City1-Ap1 1)
                           :STATUS :EXPANDED) (count 7)
FCS adding new node to open: #S(FCSN :ID NODE-1752 :K 4 :ZK (#!City1-Cl1 1)
                                     :STATUS :UNEXPANDED) (count 52)
;; ETC. (more nodes added to open)

FCS adding new node to open: #S(FCSN :ID NODE-1765 :K 4 :ZK (#!City1-Ts1 0)
                                     :STATUS :UNEXPANDED) (count 65)

FCS choosing #!PVAR-L11C = (#!City1-Ap2 1) by selecting node #S(FCSN :ID
                            NODE-1708 :K 3 :ZK (#!City1-Ap2 1) :STATUS
                            :UNEXPANDED) from FCS Open List.

CHECK-FORWARD: k=4, var=#!PVAR-L11A

REVISE: i=4 (#!PVAR-L11A), j=3 (#!PVAR-L11C) dom(i)=((#!City3-Cl1 0)
        (#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 1) (#!Region1-Ap1 0)
        (#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0)
        (#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0)
        (#!City1-Ts1 0)), dom(j)=((#!City1-Ap2 1))

CHECK-BINARY: #!PVAR-L11A and #!PVAR-L11C
```

```
FCS expanding node #S(FCSN :ID NODE-1708 :K 3 :ZK (#!City1-Ap2 1)
                           :STATUS :EXPANDED) (count 8)

FCS adding new node to open: #S(FCSN :ID NODE-1766 :K 4 :ZK (#!City1-Cl1 1)
                                     :STATUS :UNEXPANDED) (count 66)

;; etc.
;; more nodes added, one for each value in the current domain of PVAR-L11A

FCS adding new node to open: #S(FCSN :ID NODE-1779 :K 4 :ZK (#!City1-Ts1 0)
                                     :STATUS :UNEXPANDED) (count 79)

FCS choosing #!PVAR-L11C = (#!City1-Cl2 1) by selecting node #S(FCSN :ID
                            NODE-1709 :K 3 :ZK (#!City1-Cl2 1) :STATUS
                            :UNEXPANDED) from FCS Open List.

CHECK-FORWARD: k=4, var=#!PVAR-L11A

REVISE: i=4 (#!PVAR-L11A), j=3 (#!PVAR-L11C) dom(i)=((#!City3-Cl1 0)
        (#!City2-Cl1 0) (#!City1-Cl2 0) (#!City1-Cl1 1) (#!Region1-Ap1 0)
        (#!City3-Ap1 0) (#!City2-Ap1 0) (#!City1-Ap2 0) (#!City1-Ap1 0)
        (#!Region1-Ts1 0) (#!City3-Ts1 0) (#!City2-Ts1 0) (#!City1-Ts2 0)
        (#!City1-Ts1 0)), dom(j)=((#!City1-Cl2 1))

CHECK-BINARY: #!PVAR-L11A and #!PVAR-L11C

FCS expanding node #S(FCSN :ID NODE-1709 :K 3 :ZK (#!City1-Cl2 1)
                           :STATUS :EXPANDED) (count 9)
FCS adding new node to open: #S(FCSN :ID NODE-1780 :K 4 :ZK (#!City1-Cl1 1)
                                     :STATUS :UNEXPANDED) (count 80)

FCS adding new node to open: #S(FCSN :ID NODE-1781 :K 4 :ZK (#!City3-Cl1 0)
                                     :STATUS :UNEXPANDED) (count 81)

;; etc. (more nodes added to open list)

FCS adding new node to open: #S(FCSN :ID NODE-1793 :K 4 :ZK (#!City1-Ts1 0)
                                     :STATUS :UNEXPANDED) (count 93)

FCS choosing #!PVAR-L11A = (#!City1-Cl1 1) by selecting node #S(FCSN :ID
                            NODE-1710 :K 4 :ZK (#!City1-Cl1 1) :STATUS
                            :UNEXPANDED) from FCS Open List.

FCS Succeeded with node #S(FCSN :ID NODE-1710 :K 4 :ZK (#!City1-Cl1 1)
                                :STATUS :SUCCESS)
    Total Nodes Expanded = 9, Generated = 93, Total Queries = 11
Done with MAP-PLAN
PARKA (PACK) Current Query Counts (Details Collected=T):
 Total Queries: 1220
 1 Var Queries: 634
 2 Var Queries: 6
```

```
 Other Queries: 580
 Forms / Query: 1.05
 Vars  / Query: 0.54

 Call-SM calls: 13

;; The Plan Mapper is done.
;; The Plan Instantiater now creates a target plan using the newly found
;; mappings.
;; Actions and (sub)plans are added as new nodes to an initially empty HTN.
;; Note: nodes are shown as (target-id source-frame)

<<< Instantiating target plan for problem TOP-PROB-1564 from source
plan #!Plan-CE-Located-At-1.2...

Instantiated target plan TPLAN-1801.
Target Plan TPLAN-1801 (Source: #!Plan-CE-Located-At-1.2)

Actions:
TACT-1870 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.1.
TACT-1867 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.2.
TACT-1864 - Type: #!P-Collect-Fees, Roles: ((#!package= #!Pkg-1.3)),
               Source: #!P-Collect-Fees-1.1.
TACT-1883 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.3.
TACT-1892 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Truck-3)),
   Source: #!P-Open-Door-1.1.
TACT-1897 - Type: #!P-Load-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Truck-3)), Source: #!P-Load-Package-1.1.
TACT-1900 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Truck-3)),
   Source: #!P-Close-Door-1.1.
TACT-1909 - Type: #!P-Move-Vehicle, Roles: ((#!vehicle= #!Truck-3)
   (#!origin= #!City1-Cl1) (#!destin= #!City1-Cl2) (#!route=
   #!%P-Local-Road-Route)), Source: #!P-Move-Vehicle-1.1.
TACT-1920 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Truck-3)),
   Source: #!P-Open-Door-1.2.
TACT-1923 - Type: #!P-Unload-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Truck-3)), Source: #!P-Unload-Package-1.1.
TACT-1926 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Truck-3)),
   Source: #!P-Close-Door-1.2.
TACT-1929 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.4.
TACT-1932 - Type: #!P-Deliver, Roles: ((#!package= #!Pkg-1.3)),
   Source: #!P-Deliver-1.1.

PARKA (PACK) Current Query Counts (Details Collected=T):
 Total Queries: 2314
 1 Var Queries: 1174
 2 Var Queries: 6
 Other Queries: 1134
 Forms / Query: 1.02
 Vars  / Query: 0.52
```

```
 Call-SM calls: 13

;; Now the Plan Checker does annotation verification, looking for
;; applicability failures

Checking for unnecessary goals in (TPLAN-1801 #!Plan-CE-Located-At-1.2)
Checking for extra goals in TOP-PROB-1564
Checking for out facts in TOP-PROB-1564
Checking for out facts in target plan TPLAN-1801.

Checking validation [PVDTN :ID TVDTN-1802 :SOURCE #!Plan-Vdtn-1.2
:TARGET NIL :CONTRIB (GOAL-TPNODE-1797
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1897
#!P-Load-Package-1.1) :COND [PCE :ID TCOND-1803 :SFRAME
#!Plan-Vdtn-1.2-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Located-At :MODAL T :ROLES ((#!thing= [TPVAR :NAME "PVAR-T1.3"
:TVAL #!Truck-3 :SVAR #!PVAR-T1 :SVAL #!Truck-1 :CONS NIL])
(#!location= [TPVAR :NAME "PVAR-L11A.3" :TVAL #!City1-Cl1 :SVAR
#!PVAR-L11A :SVAL #!City1-Cl1 :CONS NIL]))]].

<<< Calling Parka Structure Matcher with query:
(:AND (#!instanceOf ?SITVAL #!Situated-Value) (#!location-sit=
#!Truck-3 ?SITVAL) (#!sslot-Val= ?SITVAL #!City1-Cl1) (#!situation=
?SITVAL #!Case-Make-Deliveries-3-InitSit)) <<< PSM query 14 result is
(1 bindings): (((?SITVAL . #!SitVal-1566)))

QUERY-VCON Result: (((#!PVAR-P1 . #!Pkg-1.3) (#!PVAR-L12B
. #!City1-Cl2) (#!PVAR-C1B . #!City1) (#!PVAR-T1 . #!Truck-3)
(#!PVAR-RP . #!Liquid-Package) (#!PVAR-RT . #!Tanker-Truck)
(#!PVAR-L11C . #!City1-Cl1) (#!PVAR-C1A . #!City1) (#!PVAR-L11B
. #!City1-Cl1) (#!PVAR-L11A . #!City1-Cl1) (#!PVAR-L12A)))


Checking validation [PVDTN :ID TVDTN-1804 :SOURCE #!Plan-Vdtn-1.28
:TARGET NIL :CONTRIB (GOAL-TPNODE-1797
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1920
#!P-Open-Door-1.2) :COND [PCE :ID TCOND-1805 :SFRAME
#!Plan-Vdtn-1.28-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL T :ROLES ((#!thing= [TPVAR :NAME "PVAR-T1.3"
:TVAL #!Truck-3 :SVAR #!PVAR-T1 :SVAL #!Truck-1 :CONS NIL]) (#!type=
#!Regular-Truck))]].

<<< Calling Parka Structure Matcher with query:
(#!everyInstanceOf #!Truck-3 #!Regular-Truck)
<<< PSM query 15 result is (0 bindings):
NIL

*** Failing VCON #S(VCON :ID VCON-1.28 :STATUS :CHECKED :SOLVE-P NIL
:VDTN #!Plan-Vdtn-1.28 :VARS (#!PVAR-T1) :CFORM (Q-HAS-TYPE #!PVAR-T1
#!Regular-Truck) :CMODALITY T :QFORM (Q-HAS-TYPE #!Truck-3
#!Regular-Truck) :QSUBS NIL :SIT NIL) Query forms: ((Q-HAS-TYPE
```

```
#!Truck-3 #!Regular-Truck))

;; An out fact is found (the condition of validation 28)

*** Noting out fact #S(VCON :ID VCON-1.28 :STATUS :CHECKED :SOLVE-P
NIL :VDTN #!Plan-Vdtn-1.28 :VARS (#!PVAR-T1) :CFORM (Q-HAS-TYPE
#!PVAR-T1 #!Regular-Truck) :CMODALITY T :QFORM (Q-HAS-TYPE #!Truck-3
#!Regular-Truck) :QSUBS NIL :SIT NIL)

;; ETC. (more validations checked)

;; Another failing validation (#22) due to an out fact:

*** Failing VCON #S(VCON :ID VCON-1.22 :STATUS :CHECKED :SOLVE-P NIL
:VDTN #!Plan-Vdtn-1.22 :VARS (#!PVAR-T1) :CFORM (Q-HAS-TYPE #!PVAR-T1
#!Regular-Truck) :CMODALITY T :QFORM (Q-HAS-TYPE #!Truck-3
#!Regular-Truck) :QSUBS NIL :SIT NIL) Query forms: ((Q-HAS-TYPE
#!Truck-3 #!Regular-Truck))

*** Noting out fact #S(VCON :ID VCON-1.22 :STATUS :CHECKED :SOLVE-P
NIL :VDTN #!Plan-Vdtn-1.22 :VARS (#!PVAR-T1) :CFORM (Q-HAS-TYPE
#!PVAR-T1 #!Regular-Truck) :CMODALITY T :QFORM (Q-HAS-TYPE #!Truck-3
#!Regular-Truck) :QSUBS NIL :SIT NIL)

;; ETC. (more validations checked)

;; Repair procedure called to fix failing validation 22.

REPAIR-FAILING-FILTER-CONDITION-VALIDATION [PVDTN :ID TVDTN-1844
:SOURCE #!Plan-Vdtn-1.22 :TARGET NIL :CONTRIB (GOAL-TPNODE-1797
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1892
#!P-Open-Door-1.1) :COND [PCE :ID TCOND-1845 :SFRAME
#!Plan-Vdtn-1.22-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL T :ROLES ((#!thing= [TPVAR :NAME "PVAR-T1.3"
:TVAL #!Truck-3 :SVAR #!PVAR-T1 :SVAL #!Truck-1 :CONS NIL]) (#!type=
#!Regular-Truck))]]

;; It is pruned from the set of validations.

Removing validation [PVDTN :ID TVDTN-1844 :SOURCE #!Plan-Vdtn-1.22
:TARGET NIL :CONTRIB (GOAL-TPNODE-1797
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1892
#!P-Open-Door-1.1) :COND [PCE :ID TCOND-1845 :SFRAME
#!Plan-Vdtn-1.22-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL T :ROLES ((#!thing= [TPVAR :NAME "PVAR-T1.3"
:TVAL #!Truck-3 :SVAR #!PVAR-T1 :SVAL #!Truck-1 :CONS NIL]) (#!type=
#!Regular-Truck))]]

Replacing nodes ((TPLAN-1889 #!Plan-CE-Door-Open-1.1) (TPLAN-1894
#!Plan-CE-Package-Loaded-1.1) (TPLAN-1899 #!Plan-Not-CE-Door-Open-1.1)
(TACT-1892 #!P-Open-Door-1.1) (TACT-1897 #!P-Load-Package-1.1)
```

```
(TACT-1900 #!P-Close-Door-1.1)) with (TPLAN-1887
#!Plan-CE-Package-Loaded-1.2) in partial ordering.

;; The subplan that introduced the failing condition is removed.

Remove old reduction (splan) of node (TPLAN-1887
#!Plan-CE-Package-Loaded-1.2).

;; A plan refit task is added that will cause the deleted subplan to
;; be replaced with a new subplan that will be retrieved from memory.

Adding plan refit task [PMTASK PMTASK-1935 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-1564
:VDTN [PVDTN :ID TVDTN-1844 :SOURCE #!Plan-Vdtn-1.22]]

;; The other failing validation (#28) due to an out fact is processed in
;; a similar manner:

REPAIR-FAILING-FILTER-CONDITION-VALIDATION [PVDTN :ID TVDTN-1804
:SOURCE #!Plan-Vdtn-1.28 :TARGET NIL :CONTRIB (GOAL-TPNODE-1797
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1920
#!P-Open-Door-1.2) :COND [PCE :ID TCOND-1805 :SFRAME
#!Plan-Vdtn-1.28-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL T :ROLES ((#!thing= [TPVAR :NAME "PVAR-T1.3"
:TVAL #!Truck-3 :SVAR #!PVAR-T1 :SVAL #!Truck-1 :CONS NIL]) (#!type=
#!Regular-Truck))]]

Removing validation [PVDTN :ID TVDTN-1804 :SOURCE #!Plan-Vdtn-1.28]

Replacing nodes ((TPLAN-1917 #!Plan-CE-Door-Open-1.2) (TPLAN-1922
#!Plan-Not-CE-Package-Loaded-1.1) (TPLAN-1925
#!Plan-Not-CE-Door-Open-1.2) (TACT-1920 #!P-Open-Door-1.2) (TACT-1923
#!P-Unload-Package-1.1) (TACT-1926 #!P-Close-Door-1.2)) with
(TPLAN-1915 #!Plan-Not-CE-Package-Loaded-1.2) in partial ordering.

Remove old reduction (splan) of node (TPLAN-1915
#!Plan-Not-CE-Package-Loaded-1.2).

Adding plan refit task [PMTASK PMTASK-1936 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB
TOP-PROB-1564 :VDTN [PVDTN :ID TVDTN-1804 :SOURCE #!Plan-Vdtn-1.28]]

;; The Plan Checker has added 2 tasks to the Plan Modifier's queue.

PARKA (PACK) Current Query Counts (Details Collected=T):
 Total Queries: 2467
 1 Var Queries: 1246
 2 Var Queries: 6
 Other Queries: 1215
 Forms / Query: 1.03
 Vars  / Query: 0.51
```

```
 Call-SM calls: 38

;; The Plan Modifier processes the first task, rereducing node TPLAN-1915.

Selecting and Processing PMTASK [PMTASK PMTASK-1936 :PTYPE :REREDUCE
:STYPE :REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB
TOP-PROB-1564 :VDTN [PVDTN :ID TVDTN-1804]]

;; CAPER will create a (sub)problem description for the goal of
;; unloading Truck-3

Target Problem PROB-1949 (NIL) (Class UMT-PDESC):
          Domain: :UM-TRANSLOG ("UM-Translog Domain")
            Case: NIL,  Plan: NIL,  Prev. Plans: NIL
          RPLANs: (RPLAN-1950)
   Source PPLANS: NIL
   Target PPLANS: NIL
      Root PPLAN: NIL (NIL)
     Subproblems: NIL
   Features: (UMT-PLAN-GOAL-FEAT-1945)
   Mappings: NIL
   Srcs. Chosen: NIL

;; A new RPLAN/RDESC with the target goal feature is created.

<<< Retrieving plan using RPLAN RPLAN-1950...
Rtvl. Plan RPLAN-1950 (RPLAN)
    Derived-via: :AUTO
        Problem: PROB-1949
       Features: (UMT-PLAN-GOAL-FEAT-1945)
         RDESCs: (RDESC-1951)
         Status: NIL
    Succ. RDESC: NIL
   Sources Rtvd: NIL
   Srcs. Chosen: NIL
Rtvl. Description RDESC-1951
   Derived-from: NIL
    Derived-via: :AUTO
     Rtvl. Plan: RPLAN-1950
    Query Forms: NIL
    Query Vars.: NIL
   Query Source: NIL
   Query Status: NIL
   Query Result: NIL
   Features: (UMT-PLAN-GOAL-FEAT-1945)
   Mappings: NIL

;; The Plan Retriever is invoked to find a source plan for unloading Truck-3.

<<< Beginning RDESC search from initial rdescs: (RDESC-1951)
```

```
Current OPEN list:
((NODE-1952 RDESC-1951 NIL 0 NIL))

<<< Iteration 1:

<<< Expanding New NODE-1952: Old RDESC-1951 (node NIL), New RDESC-1951

<<< Calling Parka Structure Matcher with query:

(:AND (#!goal-of-Plan ?GOAL ?PLAN) (#!instanceOf ?PKG-PVAR
#!Plan-Variable) (#!var-Value= ?PKG-PVAR ?PKG) (#!package= ?GOAL
?PKG-PVAR) (#!instanceOf ?VEH-PVAR #!Plan-Variable) (#!var-Value=
?VEH-PVAR #!Truck-3) (#!vehicle= ?GOAL ?VEH-PVAR) (#!instanceOf
?LOC-PVAR #!Plan-Variable) (#!var-Value= ?LOC-PVAR #!City1-Cl2)
(#!location= ?GOAL ?LOC-PVAR) (#!instanceOf ?PLAN #!E-Plan)
(#!modality= ?GOAL #!False) (#!instanceOf ?GOAL #!CE-Package-Loaded)
(#!everyInstanceOf ?PKG #!Liquid-Package))
<<< PSM query 39 result is (0 bindings):
NIL
<<< New query forms were: NIL

3 Successor nodes generated for node NODE-1952.

<<< Generating New NODE-1956: Old RDESC-1951 (node NODE-1952), New NIL
      Via New RDOP-1953 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 0,
      Cost 1] Variablize FACET-1942 (UMT-FCT-LOC) of
      UMT-PLAN-GOAL-FEAT-1945 Facet Var Types: ((#!everyInstanceOf NIL
      (#!City-Location)))

<<< Generating New NODE-1957: Old RDESC-1951 (node NODE-1952), New NIL
      Via New RDOP-1954 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 2,
      Cost 1] Variablize FACET-1943 (UMT-FCT-VEH) of
      UMT-PLAN-GOAL-FEAT-1945 Facet Var Types: NIL

<<< Generating New NODE-1958: Old RDESC-1951 (node NODE-1952), New NIL
      Via New RDOP-1955 (RDOP-GENERALIZE-FACET) [Status NIL, Score 3,
      Cost 1] Replace Type FVT-1948 ((#!everyInstanceOf
      #!Liquid-Package (#!Liquid-Package #!Package))) of FACET-1944
      with: ((#!everyInstanceOf #!Package (#!Liquid-Package
      #!Package)))

Current OPEN list:
((NODE-1956 RDESC-1951 RDOP-1953 1 0) (NODE-1957 RDESC-1951 RDOP-1954
1 2) (NODE-1958 RDESC-1951 RDOP-1955 1 3))

<<< Iteration 2:

<<< Expanding New NODE-1956: Old RDESC-1951 (node NODE-1952), New NIL
      Via New RDOP-1953 (RDOP-VARIABILIZE-FACET) [Status 2, Score 0, Cost 1]
    Variablize FACET-1942 (UMT-FCT-LOC) of UMT-PLAN-GOAL-FEAT-1945
```

```
    Facet Var Types: ((#!everyInstanceOf NIL (#!City-Location)))

<<< Calling Parka Structure Matcher with query:
(:AND (#!goal-of-Plan ?GOAL ?PLAN) (#!instanceOf ?PKG-PVAR
 #!Plan-Variable) (#!var-Value= ?PKG-PVAR ?PKG) (#!package= ?GOAL
 ?PKG-PVAR) (#!instanceOf ?VEH-PVAR #!Plan-Variable) (#!var-Value=
 ?VEH-PVAR #!Truck-3) (#!vehicle= ?GOAL ?VEH-PVAR) (#!instanceOf
 ?LOC-PVAR #!Plan-Variable) (#!var-Value= ?LOC-PVAR ?LOC) (#!location=
 ?GOAL ?LOC-PVAR) (#!everyInstanceOf ?PKG #!Liquid-Package)
 (#!instanceOf ?GOAL #!CE-Package-Loaded) (#!modality= ?GOAL #!False)
 (#!instanceOf ?PLAN #!E-Plan) (#!everyInstanceOf ?LOC
 #!City-Location))
<<< PSM query 40 result is (0 bindings): NIL
<<< New query forms were: ((#!everyInstanceOf ?LOC #!City-Location)
 (#!var-Value= ?LOC-PVAR ?LOC))

2 Successor nodes generated for node NODE-1956.

<<< Generating New NODE-1976: Old RDESC-1973 (node NODE-1956), New NIL
      Via New RDOP-1974 (RDOP-GENERALIZE-FACET) [Status NIL, Score 3,
      Cost 2] Replace Type FVT-1970 ((#!everyInstanceOf
      #!Liquid-Package (#!Liquid-Package #!Package))) of FACET-1971
      with: ((#!everyInstanceOf #!Package (#!Liquid-Package
      #!Package)))

<<< Generating New NODE-1977: Old RDESC-1973 (node NODE-1956), New NIL
      Via New RDOP-1975 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 2,
      Cost 2] Variablize FACET-1969 (UMT-FCT-VEH) of
      UMT-PLAN-GOAL-FEAT-1972 Facet Var Types: NIL

Current OPEN list:
((NODE-1957 RDESC-1951 RDOP-1954 1 2) (NODE-1977 RDESC-1973 RDOP-1975
 2 2) (NODE-1958 RDESC-1951 RDOP-1955 1 3) (NODE-1976 RDESC-1973
 RDOP-1974 2 3))

<<< Iteration 3:

<<< Expanding New NODE-1957: Old RDESC-1951 (node NODE-1952), New NIL
      Via New RDOP-1954 (RDOP-VARIABILIZE-FACET) [Status 3, Score 2, Cost 1]
    Variablize FACET-1943 (UMT-FCT-VEH) of UMT-PLAN-GOAL-FEAT-1945
    Facet Var Types: NIL

<<< Calling Parka Structure Matcher with query:
(:AND (#!goal-of-Plan ?GOAL ?PLAN) (#!instanceOf ?PKG-PVAR
 #!Plan-Variable) (#!var-Value= ?PKG-PVAR ?PKG) (#!package= ?GOAL
 ?PKG-PVAR) (#!instanceOf ?VEH-PVAR #!Plan-Variable) (#!var-Value=
 ?VEH-PVAR ?VEH) (#!vehicle= ?GOAL ?VEH-PVAR) (#!instanceOf ?LOC-PVAR
 #!Plan-Variable) (#!var-Value= ?LOC-PVAR #!City1-Cl2) (#!location=
 ?GOAL ?LOC-PVAR) (#!everyInstanceOf ?PKG #!Liquid-Package)
 (#!instanceOf ?GOAL #!CE-Package-Loaded) (#!modality= ?GOAL #!False)
 (#!instanceOf ?PLAN #!E-Plan) (#!instanceOf ?VEH #!Tanker-Truck))
```

```
<<< PSM query 41 result is (0 bindings):
NIL
<<< New query forms were: ((#!instanceOf ?VEH #!Tanker-Truck)
(#!var-Value= ?VEH-PVAR ?VEH))

2 Successor nodes generated for node NODE-1957.

<<< Generating New NODE-1996: Old RDESC-1993 (node NODE-1957), New NIL
      Via New RDOP-1994 (RDOP-GENERALIZE-FACET) [Status NIL, Score 3,
      Cost 2] Replace Type FVT-1990 ((#!everyInstanceOf
      #!Liquid-Package (#!Liquid-Package #!Package))) of FACET-1991
      with: ((#!everyInstanceOf #!Package (#!Liquid-Package
      #!Package)))

<<< Generating New NODE-1997: Old RDESC-1993 (node NODE-1957), New NIL
      Via New RDOP-1995 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 0,
      Cost 2] Variablize FACET-1989 (UMT-FCT-LOC) of
      UMT-PLAN-GOAL-FEAT-1992 Facet Var Types: ((#!everyInstanceOf NIL
      (#!City-Location)))

Current OPEN list:
((NODE-1997 RDESC-1993 RDOP-1995 2 0) (NODE-1977 RDESC-1973 RDOP-1975
 2 2) (NODE-1958 RDESC-1951 RDOP-1955 1 3) (NODE-1996 RDESC-1993
 RDOP-1994 2 3) (NODE-1976 RDESC-1973 RDOP-1974 2 3))

<<< Iteration 4:

<<< Expanding New NODE-1997: Old RDESC-1993 (node NODE-1957), New NIL
      Via New RDOP-1995 (RDOP-VARIABILIZE-FACET) [Status 4, Score 0, Cost 2]
    Variablize FACET-1989 (UMT-FCT-LOC) of UMT-PLAN-GOAL-FEAT-1992
    Facet Var Types: ((#!everyInstanceOf NIL (#!City-Location)))

<<< Calling Parka Structure Matcher with query:
(:AND (#!goal-of-Plan ?GOAL ?PLAN) (#!instanceOf ?PKG-PVAR
 #!Plan-Variable) (#!var-Value= ?PKG-PVAR ?PKG) (#!package= ?GOAL
 ?PKG-PVAR) (#!instanceOf ?VEH-PVAR #!Plan-Variable) (#!var-Value=
 ?VEH-PVAR ?VEH) (#!vehicle= ?GOAL ?VEH-PVAR) (#!instanceOf ?LOC-PVAR
 #!Plan-Variable) (#!var-Value= ?LOC-PVAR ?LOC) (#!location= ?GOAL
 ?LOC-PVAR) (#!instanceOf ?VEH #!Tanker-Truck) (#!instanceOf ?PLAN
 #!E-Plan) (#!modality= ?GOAL #!False) (#!instanceOf ?GOAL
 #!CE-Package-Loaded) (#!everyInstanceOf ?PKG #!Liquid-Package)
 (#!everyInstanceOf ?LOC #!City-Location))
<<< PSM query 42 result is (1 bindings): (((?GOAL
. #!Plan-Not-CE-Package-Loaded-2.1-Goal-1) (?PLAN
. #!Plan-Not-CE-Package-Loaded-2.1) (?LOC . #!City2-Cl1) (?LOC-PVAR
. #!PVAR-1819) (?VEH . #!Truck-4) (?VEH-PVAR . #!PVAR-1815) (?PKG
. #!Pkg-1.2) (?PKG-PVAR . #!PVAR-1814)))
<<< New query forms were: ((#!everyInstanceOf ?LOC #!City-Location)
(#!var-Value= ?LOC-PVAR ?LOC))
```

```
<<< Done with RDESC Search.  4 nodes expanded
    Search Status is SUCCESS.

PARKA (PACK) Current Query Counts (Details Collected=T):
 Total Queries: 2474
 1 Var Queries: 1247
 2 Var Queries: 10
 Other Queries: 1217
 Forms / Query: 1.05
 Vars  / Query: 0.52

 Call-SM calls: 42

;; Plan retrieval was successful.  The single source plan retrieved is
;; chosen, mapped, and instantiated.

<<< Mapping source plan #!Plan-Not-CE-Package-Loaded-2.1 to problem
PROB-1949...
Done with MAP-PLAN

<<< Instantiating target plan for problem PROB-1949 from source plan
#!Plan-Not-CE-Package-Loaded-2.1...

Instantiated target plan TPLAN-2026.

Target Plan TPLAN-2026 (Source: #!Plan-Not-CE-Package-Loaded-2.1)

Actions: TACT-2047 - Type: #!P-Connect-Hose, Roles: ((#!vehicle=
#!Truck-3) (#!package= #!Pkg-1.3)), Source: #!P-Connect-Hose-2.2.
TACT-2054 - Type: #!P-Open-Vehicle-Valve, Roles: ((#!vehicle=
#!Truck-3)), Source: #!P-Open-Vehicle-Valve-2.2.  TACT-2057 - Type:
#!P-Empty-Tank, Roles: ((#!vehicle= #!Truck-3)), Source:
#!P-Empty-Tank-2.1.  TACT-2060 - Type: #!P-Close-Vehicle-Valve, Roles:
((#!vehicle= #!Truck-3)), Source: #!P-Close-Vehicle-Valve-2.2.
TACT-2063 - Type: #!P-Disconnect-Hose, Roles: ((#!vehicle=
#!Truck-3)), Source: #!P-Disconnect-Hose-2.2.

PARKA (PACK) Current Query Counts (Details Collected=T):
 Total Queries: 2946
 1 Var Queries: 1480
 2 Var Queries: 10
 Other Queries: 1456
 Forms / Query: 1.05
 Vars  / Query: 0.52

 Call-SM calls: 42

;; The new subplan TPLAN-2026 is now checked for interactions with the
;; overall plan.

<<< Processing replacement plan (TPLAN-2026
```

```
#!Plan-Not-CE-Package-Loaded-2.1) for problem PROB-1949...
<<< Checking replacement plan (TPLAN-2026
#!Plan-Not-CE-Package-Loaded-2.1) for problem PROB-1949...

Checking E-Preconditions of (TPLAN-2026 #!Plan-Not-CE-Package-Loaded-2.1)
Checking E-Conditions of (TPLAN-1915 #!Plan-Not-CE-Package-Loaded-1.2)

;; Various unnecessary validations will be pruned later.

;; The actual merging of the subplan occurs by adding its nodes to the HTN.

Replacing node (TPLAN-1915 #!Plan-Not-CE-Package-Loaded-1.2) with
(TPLAN-2026 #!Plan-Not-CE-Package-Loaded-2.1) in HTN HTN-1798

Replacing nodes (TPLAN-1915 #!Plan-Not-CE-Package-Loaded-1.2) with
((TPLAN-2026 #!Plan-Not-CE-Package-Loaded-2.1) (TPLAN-2044
#!Plan-CE-Hose-Connected-2.2) (TACT-2047 #!P-Connect-Hose-2.2)
(TPLAN-2049 #!Plan-CE-Valve-Open-2.2) (TACT-2054
#!P-Open-Vehicle-Valve-2.2) (TPLAN-2056 #!Plan-Not-CE-Full-Tank-2.1)
(TACT-2057 #!P-Empty-Tank-2.1) (TPLAN-2059
#!Plan-Not-CE-Valve-Open-2.2) (TACT-2060 #!P-Close-Vehicle-Valve-2.2)
(TPLAN-2062 #!Plan-Not-CE-Hose-Connected-2.2) (TACT-2063
#!P-Disconnect-Hose-2.2)) in partial ordering.

;; The other rereduce task is processed.  A new subplan is found to
;; load Truck-3 and merged as above.  Much of this output for this is
;; ommitted for brevity.

Selecting and Processing PMTASK [PMTASK PMTASK-1935 :PTYPE :REREDUCE
:STYPE :REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB
TOP-PROB-1564 :VDTN [PVDTN :ID TVDTN-1844]]

;; ETC.  (output from Plan Retriever and Mapper on subplan ommitted)

Instantiated target plan TPLAN-2178.

Target Plan TPLAN-2178 (Source: #!Plan-CE-Package-Loaded-2.1)
Actions: TACT-2200 - Type: #!P-Connect-Hose, Roles: ((#!vehicle=
#!Truck-3) (#!package= #!Pkg-1.3)), Source: #!P-Connect-Hose-2.1.
TACT-2209 - Type: #!P-Open-Vehicle-Valve, Roles: ((#!vehicle=
#!Truck-3)), Source: #!P-Open-Vehicle-Valve-2.1.  TACT-2212 - Type:
#!P-Fill-Tank, Roles: ((#!vehicle= #!Truck-3)), Source:
#!P-Fill-Tank-2.1.  TACT-2215 - Type: #!P-Close-Vehicle-Valve, Roles:
((#!vehicle= #!Truck-3)), Source: #!P-Close-Vehicle-Valve-2.1.
TACT-2218 - Type: #!P-Disconnect-Hose, Roles: ((#!vehicle=
#!Truck-3)), Source: #!P-Disconnect-Hose-2.1.

PARKA (PACK) Current Query Counts (Details Collected=T):
 Total Queries: 3557
 1 Var Queries: 1790
 2 Var Queries: 15
```

```
 Other Queries: 1752
 Forms / Query: 1.05
 Vars  / Query: 0.53

 Call-SM calls: 47
<<< Processing replacement plan (TPLAN-2178
#!Plan-CE-Package-Loaded-2.1) for problem PROB-2086...
<<< Checking replacement plan (TPLAN-2178
#!Plan-CE-Package-Loaded-2.1) for problem PROB-2086...

;; ETC.
;; New load subplan is merged into overall target plan.

Replacing node (TPLAN-1887 #!Plan-CE-Package-Loaded-1.2) with
(TPLAN-2178 #!Plan-CE-Package-Loaded-2.1) in HTN HTN-1798

Replacing nodes (TPLAN-1887 #!Plan-CE-Package-Loaded-1.2) with
((TPLAN-2178 #!Plan-CE-Package-Loaded-2.1) (TPLAN-2197
#!Plan-CE-Hose-Connected-2.1) (TACT-2200 #!P-Connect-Hose-2.1)
(TPLAN-2202 #!Plan-CE-Valve-Open-2.1) (TACT-2209
#!P-Open-Vehicle-Valve-2.1) (TPLAN-2211 #!Plan-CE-Full-Tank-2.1)
(TACT-2212 #!P-Fill-Tank-2.1) (TPLAN-2214
#!Plan-Not-CE-Valve-Open-2.1) (TACT-2215 #!P-Close-Vehicle-Valve-2.1)
(TPLAN-2217 #!Plan-Not-CE-Hose-Connected-2.1) (TACT-2218
#!P-Disconnect-Hose-2.1)) in partial ordering.

;; ETC.
;; Remaining prune tasks are processed.  Resulting plan is:

Target Plan TPLAN-1801 (Source: #!Plan-CE-Located-At-1.2) Actions:
TACT-1870 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.1.
TACT-1867 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.2.
TACT-1864 - Type: #!P-Collect-Fees, Roles: ((#!package= #!Pkg-1.3)),
   Source: #!P-Collect-Fees-1.1.
TACT-1883 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.3.
TACT-2200 - Type: #!P-Connect-Hose, Roles: ((#!vehicle= #!Truck-3)
   (#!package= #!Pkg-1.3)), Source: #!P-Connect-Hose-2.1.
TACT-2209 - Type: #!P-Open-Vehicle-Valve, Roles: ((#!vehicle=
   #!Truck-3)), Source: #!P-Open-Vehicle-Valve-2.1.
TACT-2212 - Type: #!P-Fill-Tank, Roles: ((#!vehicle= #!Truck-3)),
   Source: #!P-Fill-Tank-2.1.
TACT-2215 - Type: #!P-Close-Vehicle-Valve, Roles: ((#!vehicle=
   #!Truck-3)), Source: #!P-Close-Vehicle-Valve-2.1.
TACT-2218 - Type: #!P-Disconnect-Hose, Roles: ((#!vehicle=
   #!Truck-3)), Source: #!P-Disconnect-Hose-2.1.
TACT-1909 - Type: #!P-Move-Vehicle, Roles: ((#!vehicle= #!Truck-3)
   (#!origin= #!City1-Cl1) (#!destin= #!City1-Cl2) (#!route=
   #!%P-Local-Road-Route)), Source: #!P-Move-Vehicle-1.1.
TACT-2047 - Type: #!P-Connect-Hose, Roles: ((#!vehicle= #!Truck-3)
   (#!package= #!Pkg-1.3)), Source: #!P-Connect-Hose-2.2.
TACT-2054 - Type: #!P-Open-Vehicle-Valve, Roles: ((#!vehicle=
```

```
   #!Truck-3)), Source: #!P-Open-Vehicle-Valve-2.2.
TACT-2057 - Type: #!P-Empty-Tank, Roles: ((#!vehicle= #!Truck-3)),
   Source: #!P-Empty-Tank-2.1.
TACT-2060 - Type: #!P-Close-Vehicle-Valve, Roles: ((#!vehicle=
   #!Truck-3)), Source: #!P-Close-Vehicle-Valve-2.2.
TACT-2063 - Type: #!P-Disconnect-Hose, Roles: ((#!vehicle=
   #!Truck-3)), Source: #!P-Disconnect-Hose-2.2.
TACT-1929 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.4.
TACT-1932 - Type: #!P-Deliver, Roles: ((#!package= #!Pkg-1.3)),
   Source: #!P-Deliver-1.1.

;; New plan is added as a case in the casebase.

<<< Storing target plan TPLAN-1801 and problem PROB-2086 as new case
#!Case-Make-Deliveries-3...

Created 167 new frames and 3531 new assertions for case
#!Case-Make-Deliveries-3.

Action frames for plan #!Plan-CE-Located-At-3.1: (#!Do-Nothing-3.2
#!Do-Nothing-3.1 #!P-Collect-Fees-3.1 #!Do-Nothing-3.3
#!P-Connect-Hose-3.1 #!P-Open-Vehicle-Valve-3.1 #!P-Fill-Tank-3.1
#!P-Close-Vehicle-Valve-3.1 #!P-Disconnect-Hose-3.1
#!P-Move-Vehicle-3.1 #!P-Connect-Hose-3.2 #!P-Open-Vehicle-Valve-3.2
#!P-Empty-Tank-3.1 #!P-Close-Vehicle-Valve-3.2 #!P-Disconnect-Hose-3.2
#!Do-Nothing-3.4 #!P-Deliver-3.1)

New plan for case #!Case-Make-Deliveries-3:

CAPER has finished. Termination status = :SUCCESS.
```

7.2 Example 2: Delivering a Hazardous Package between Cities with a Hub

This more complicated example is presented in abbreviated mode with the focus being on additional capabilities of CAPER. The target problem is to deliver a hazardous regular-type package from City1 to City2, where these cities are connected by train via hubs City3 or City4. The former hub does not permit hazardous packages. CAPER initially finds a source plan that transports a nonhazardous regular-type package through City3. During the mapping of this plan to the target situation, it notices (via the plan variable constraints arising from the source plan validations) that it must choose an alternate hub, City4. Thus City3 (and its connecting routes to City1 and City2) is replaced with City4. The resulting target plan must be modified to use the appropriate loading/unloading methods for hazardous packages (involving the affixing/removal of warning signs) and to obtain a permit prior to transport. These modifications are done by retrieving and merging subplans from a second source case in which a hazardous package is transported. Ultimately 6 source (sub)plans drawn from two source cases will be used (adapted and merged).

```
THE CAPER CASE-BASED PLANNING SYSTEM
(Version 0.3.1.)
(c) 1995, Dept. of Computer Science
University of Maryland at College Park

UMT-PDESC Problem to be solved: IRR-HR-1-2 (World: WORLD-RT2)

New problem is designated as case "Case-Make-Deliveries-3".
Defining new frames for case "Case-Make-Deliveries-3" objects...
Defining plan frame "Plan-Make-Deliveries-3"
Initializing world frames for situation #!Case-Make-Deliveries-3-InitSit

Defining initial features for case "Case-Make-Deliveries-3" objects...
Generated initial problem description (*target-pdesc*):
Target Problem TOP-PROB-494 (IRR-HR-1-2) (Class UMT-PDESC):
          Domain: :UM-TRANSLOG ("UM-Translog Domain")
           Case: #!Case-Make-Deliveries-3,  Plan: #!Plan-Make-Deliveries-3,
            Prev. Plans: NIL
          RPLANs: NIL
   Source PPLANS: NIL
   Target PPLANS: NIL
      Root PPLAN: NIL (#!Plan-Make-Deliveries-3)
     Subproblems: NIL
   Features: (UMT-GOAL-TRANSPORT-PKG-509)
   Mappings: NIL
Feature UMT-GOAL-TRANSPORT-PKG-509 (prob.fid: UMT-GOAL-TRANSPORT-PKG-509):
            Class: UMT-GOAL-TRANSPORT-PKG
       Feature Of: :TARGET,  Derived From: NIL,  Derived Via: :PREDEFINED
       Facets: (FACET-511 FACET-512 FACET-513 FACET-515 FACET-517 FACET-519)
   Facet bindings:
((PLAN #!Plan-Make-Deliveries-3) (GOAL #!Case-Make-Deliveries-3-Goal-1) (PKG
 #!Pkg-1.3) (ORGN #!City1-Ts) (DEST #!City2-Ts) (ISIT
 #!Case-Make-Deliveries-3-InitSit))
New RPLAN RPLAN-523 created for target problem.

;; The retrieval probe is generalized until a plan for delivering
;; a regular package between the target origin and target destination
;; is found...

<<< Retrieving plan using RPLAN RPLAN-523...
Rtvl. Plan RPLAN-523 (RPLAN)
    Derived-via: :AUTO
        Problem: TOP-PROB-494
       Features: (UMT-GOAL-TRANSPORT-PKG-509)
         RDESCs: (RDESC-524)
         Status: NIL
    Succ. RDESC: NIL
   Sources Rtvd: NIL
   Srcs. Chosen: NIL
```

```
Rtvl. Description RDESC-524
   Derived-from: NIL
    Derived-via: :AUTO
     Rtvl. Plan: RPLAN-523
    Query Forms: NIL
    Query Vars.: NIL
   Query Source: NIL
   Query Status: NIL
   Query Result: NIL
   Features: (UMT-GOAL-TRANSPORT-PKG-509)
   Mappings: NIL

<<< Beginning RDESC search from initial rdescs: (RDESC-524)
      Parameters: Query Mode       (*rds-query-mode*): T
                  Node Succs Fn (*rds-node-succs-fn*): NIL
                  Node Score Fn (*rds-node-score-fn*):
                  #<Compiled-function GET-RDSN-BASE-SCORE #x7847F6>
                  Order Open Fn (*rds-order-open-fn*):
                  #<Compiled-function RDS-ORDER-OPEN-BEST #x785ECE>
                  Eval Match Fn (*rds-eval-match-fn*):
                  #<Compiled-function NEW-PLANS-FOUND #x78798E>
                  Ignore Seen States? (*rds-ignore-seen-states-p*): T

Current OPEN list:
((NODE-525 RDESC-524 NIL 0 NIL))

<<< Iteration 1:

<<< Expanding New NODE-525: Old RDESC-524 (node NIL), New RDESC-524

<<< Calling Parka Structure Matcher with query:
(:AND (#!instanceOf ?PKG-PVAR #!Plan-Variable) (#!var-Value= ?PKG-PVAR ?PKG)
 (#!thing= ?GOAL ?PKG-PVAR) (#!instanceOf ?DEST-PVAR #!Plan-Variable)
 (#!var-Value= ?DEST-PVAR #!City2-Ts) (#!location= ?GOAL ?DEST-PVAR)
 (#!goal-of-Plan ?GOAL ?PLAN) (#!situation-Before= ?PLAN ?ISIT)
 (#!instanceOf ?SITVAL #!Situated-Value) (#!location-sit= ?PKG ?SITVAL)
 (#!sslot-Val= ?SITVAL #!City1-Ts) (#!situation= ?SITVAL ?ISIT)
 (#!instanceOf ?PLAN #!E-Plan) (#!modality= ?GOAL #!True) (#!instanceOf
 ?GOAL #!CE-Located-At) (#!everyInstanceOf ?PKG #!Hazardous-Regular-Package)
 (#!instanceOf ?ISIT #!Initial-Situation))
<<< PSM query 1 result is: NIL
<<< New query forms were: NIL

3 Successor nodes generated for node NODE-525.

<<< Generating New NODE-529: Old RDESC-524 (node NODE-525), New NIL Via New
      RDOP-526 (RDOP-GENERALIZE-FACET) [Status NIL, Score 0, Cost 1] Replace
      Type FVT-522 ((#!everyInstanceOf #!Hazardous-Regular-Package
      (#!Hazardous-Regular-Package #!Hazardous-Package #!Regular-Package
      #!Package))) of FACET-513 with: ((#!everyInstanceOf
      #!Hazardous-Package (#!Hazardous-Regular-Package #!Hazardous-Package
```

```
      #!Regular-Package #!Package)))

<<< Generating New NODE-530: Old RDESC-524 (node NODE-525), New NIL Via New
      RDOP-527 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 1, Cost 1]
      Variablize FACET-515 (UMT-FCT-LOC-GEO) of UMT-GOAL-TRANSPORT-PKG-509
      Facet Var Types: ((#!everyInstanceOf NIL (#!City-Location)))

<<< Generating New NODE-531: Old RDESC-524 (node NODE-525), New NIL Via New
      RDOP-528 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 1, Cost 1]
      Variablize FACET-517 (UMT-FCT-LOC-GEO) of UMT-GOAL-TRANSPORT-PKG-509
      Facet Var Types: ((#!everyInstanceOf NIL (#!City-Location)))

Current OPEN list:
((NODE-529 RDESC-524 RDOP-526 1 0) (NODE-531 RDESC-524 RDOP-528 1 1)
(NODE-530 RDESC-524 RDOP-527 1 1))

<<< Iteration 2:

<<< Expanding New NODE-529: Old RDESC-524 (node NODE-525), New NIL Via New
      RDOP-526 (RDOP-GENERALIZE-FACET) [Status 2, Score 0, Cost 1] Replace
      Type FVT-522 ((#!everyInstanceOf #!Hazardous-Regular-Package
      (#!Hazardous-Regular-Package #!Hazardous-Package #!Regular-Package
      #!Package))) of FACET-513 with: ((#!everyInstanceOf
      #!Hazardous-Package (#!Hazardous-Regular-Package #!Hazardous-Package
      #!Regular-Package #!Package)))

<<< Calling Parka Structure Matcher with query:
(:AND (#!instanceOf ?PKG-PVAR #!Plan-Variable) (#!var-Value= ?PKG-PVAR ?PKG)
 (#!thing= ?GOAL ?PKG-PVAR) (#!instanceOf ?DEST-PVAR #!Plan-Variable)
 (#!var-Value= ?DEST-PVAR #!City2-Ts) (#!location= ?GOAL ?DEST-PVAR)
 (#!goal-of-Plan ?GOAL ?PLAN) (#!situation-Before= ?PLAN ?ISIT)
 (#!instanceOf ?SITVAL #!Situated-Value) (#!location-sit= ?PKG ?SITVAL)
 (#!sslot-Val= ?SITVAL #!City1-Ts) (#!situation= ?SITVAL ?ISIT)
 (#!instanceOf ?ISIT #!Initial-Situation) (#!everyInstanceOf ?PKG
 #!Hazardous-Package) (#!instanceOf ?GOAL #!CE-Located-At) (#!modality=
 ?GOAL #!True) (#!instanceOf ?PLAN #!E-Plan))
<<< PSM query 2 result is: NIL
<<< New query forms were: ((#!everyInstanceOf ?PKG #!Hazardous-Package))

3 Successor nodes generated for node NODE-529.

<<< Generating New NODE-565: Old RDESC-561 (node NODE-529), New NIL Via New
      RDOP-562 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 1, Cost 2]
      Variablize FACET-545 (UMT-FCT-LOC-GEO) of UMT-GOAL-TRANSPORT-PKG-550
      Facet Var Types: ((#!everyInstanceOf NIL (#!City-Location)))

<<< Generating New NODE-566: Old RDESC-561 (node NODE-529), New NIL Via New
      RDOP-563 (RDOP-VARIABILIZE-FACET) [Status NIL, Score 1, Cost 2]
      Variablize FACET-542 (UMT-FCT-LOC-GEO) of UMT-GOAL-TRANSPORT-PKG-550
      Facet Var Types: ((#!everyInstanceOf NIL (#!City-Location)))
```

```
<<< Generating New NODE-567: Old RDESC-561 (node NODE-529), New NIL Via New
      RDOP-564 (RDOP-GENERALIZE-FACET) [Status NIL, Score 0, Cost 2] Replace
      Type FVT-522 ((#!everyInstanceOf #!Hazardous-Package
      (#!Hazardous-Regular-Package #!Hazardous-Package #!Regular-Package
      #!Package))) of FACET-539 with: ((#!everyInstanceOf #!Regular-Package
      (#!Hazardous-Regular-Package #!Hazardous-Package #!Regular-Package
      #!Package)))

Current OPEN list:
((NODE-567 RDESC-561 RDOP-564 2 0) (NODE-531 RDESC-524 RDOP-528 1 1)
(NODE-53 0 RDESC-524 RDOP-527 1 1) (NODE-566 RDESC-561 RDOP-563 2 1)
(NODE-565 RDESC-561 RDOP-562 2 1))

<<< Iteration 3:

<<< Expanding New NODE-567: Old RDESC-561 (node NODE-529), New NIL Via New
      RDOP-564 (RDOP-GENERALIZE-FACET) [Status 3, Score 0, Cost 2] Replace
      Type FVT-522 ((#!everyInstanceOf #!Hazardous-Package
      (#!Hazardous-Regular-Package #!Hazardous-Package #!Regular-Package
      #!Package))) of FACET-539 with: ((#!everyInstanceOf #!Regular-Package
      (#!Hazardous-Regular-Package #!Hazardous-Package #!Regular-Package
      #!Package)))

<<< Calling Parka Structure Matcher with query:
(:AND (#!instanceOf ?PKG-PVAR #!Plan-Variable) (#!var-Value= ?PKG-PVAR ?PKG)
 (#!thing= ?GOAL ?PKG-PVAR) (#!instanceOf ?DEST-PVAR #!Plan-Variable)
 (#!var-Value= ?DEST-PVAR #!City2-Ts) (#!location= ?GOAL ?DEST-PVAR)
 (#!goal-of-Plan ?GOAL ?PLAN) (#!situation-Before= ?PLAN ?ISIT)
 (#!instanceOf ?SITVAL #!Situated-Value) (#!location-sit= ?PKG ?SITVAL)
 (#!sslot-Val= ?SITVAL #!City1-Ts) (#!situation= ?SITVAL ?ISIT)
 (#!instanceOf ?PLAN #!E-Plan) (#!modality= ?GOAL #!True) (#!instanceOf
 ?GOAL #!CE-Located-At) (#!everyInstanceOf ?PKG #!Regular-Package)
 (#!instanceOf ?ISIT #!Initial-Situation))
<<< PSM query 3 result is (2 bindings):
(((?GOAL . #!Plan-CE-Located-At-1.5-Goal-1) (?PLAN
  . #!Plan-CE-Located-At-1.5) (?PKG-PVAR . #!PVAR-1440) (?SITVAL
  . #!SitVal-1439) (?PKG . #!Pkg-1.1) (?DEST-PVAR . #!PVAR-1449) (?ISIT
  . #!Case-Make-Deliveries-1-InitSit)) ((?GOAL
  . #!Plan-CE-Located-At-1.6-Goal-1) (?PLAN . #!Plan-CE-Located-At-1.6)
  (?PKG-PVAR . #!PVAR-1440) (?SITVAL . #!SitVal-1439) (?PKG . #!Pkg-1.1)
  (?DEST-PVAR . #!PVAR-1449) (?ISIT . #!Case-Make-Deliveries-1-InitSit)))
<<< New query forms were: ((#!everyInstanceOf ?PKG #!Regular-Package))

<<< Done with RDESC Search.  3 nodes expanded Search Status is
    SUCCESS.
Found goal node [NODE-567: (status 3) (rdesc-old RDESC-561) (op
    RDOP-564) (rdesc-new RDESC-597) (score 0) (cost 2) (parent NODE-529)
    (children NIL)] RDESC RDESC-597: Query Status :SUCCESS Feature
    UMT-GOAL-TRANSPORT-PKG-586 (orig UMT-GOAL-TRANSPORT-PKG-509): Facet
    Bindings ((PLAN #!Plan-Make-Deliveries-3) (GOAL
    #!Case-Make-Deliveries-3-Goal-1) (PKG #!Pkg-1.3) (ORGN #!City1-Ts) (DEST
```

```
    #!City2-Ts) (ISIT #!Case-Make-Deliveries-3-InitSit))

Final OPEN list: ((NODE-566 RDESC-561 RDOP-563 2 1) (NODE-565
RDESC-561 RDOP-562 2 1) (NODE-531 RDESC-524 RDOP-528 1 1) (NODE-530
RDESC-524 RDOP-527 1 1))

Final CLOSED list: ((NODE-525 RDESC-524 NIL 0 NIL) (NODE-529 RDESC-524
RDOP-526 1 0) (NODE-567 RDESC-561 RDOP-564 2 0))

<<< Retrieved 2 candidate source plans: (#!Plan-CE-Located-At-1.5
   #!Plan-CE-Located-At-1.6)

Ignoring Plan #!Plan-CE-Located-At-1.6, a superplan of
#!Plan-CE-Located-At-1.5.

<<< Selecting high scoring plan "Plan-CE-Located-At-1.5" (SDESC SPLAN-641),
         score = NIL: Features: (UMT-GOAL-TRANSPORT-PKG-598) Mappings:
         ((UMT-GOAL-TRANSPORT-PKG-509 . UMT-GOAL-TRANSPORT-PKG-598))

Actions of source plan #!Plan-CE-Located-At-1.5: (#!Do-Nothing-1.1
#!Do-Nothing-1.2 #!P-Collect-Fees-1.1 #!Do-Nothing-1.3 #!P-Open-Door-1.1
#!P-Load-Package-1.1 #!P-Close-Door-1.1 #!P-Attach-Traincar-1.1
#!P-Move-Vehicle-1.1 #!P-Detach-Traincar-1.1 #!P-Open-Door-1.2
#!P-Unload-Package-1.1 #!P-Close-Door-1.2 #!Do-Nothing-1.4 #!Do-Nothing-1.5
#!P-Open-Door-1.3 #!P-Load-Package-1.2 #!P-Close-Door-1.3
#!P-Attach-Traincar-1.2 #!P-Move-Vehicle-1.2 #!P-Detach-Traincar-1.2
#!P-Open-Door-1.4 #!P-Unload-Package-1.2 #!P-Close-Door-1.4 #!Do-Nothing-1.6
#!P-Deliver-1.1)

;; In the process of mapping, the old hub (City3-Ts) through
;; which the package was carried is noticed to be inappropriate
;; in the target situation, where a HAZARDOUS package is being
;; delivered. Thus the old hub and routes will be mapped to a
;; new hub (City4-Ts) and routes...

<<< Mapping source plan #!Plan-CE-Located-At-1.5 to problem TOP-PROB-494...

Checking hard unary vconstraint for #!PVAR-1446:

#S(VCON :ID VCON-1.10 :STATUS NIL :SOLVE-P T :VDTN #!Plan-Vdtn-1.10 :VARS
(#!PVAR-1446) :CFORM (Q-AVAILABLE #!PVAR-1446 #!True
#!Case-Make-Deliveries-3-InitSit) :CMODALITY T :QFORM (Q-AVAILABLE ?QVAR-642
#!True #!Case-Make-Deliveries-3-InitSit) :QSUBS ((?QVAR-642 . #!PVAR-1446))
:SIT :INITIAL :AVARS (#!PVAR-1446)).

<<< Calling Parka Structure Matcher with query:
(:AND (#!everyInstanceOf ?QVAR-642 #!Vehicle) (#!instanceOf ?SITVAL
 #!Situated-Value) (#!available-sit? ?QVAR-642 ?SITVAL) (#!sslot-Val=
 ?SITVAL #!True) (#!situation= ?SITVAL #!Case-Make-Deliveries-3-InitSit))
<<< PSM query 4 result is (2 bindings):
(((?QVAR-642 . #!Train-1) (?SITVAL . #!SitVal-507)) ((?QVAR-642
```

```
. #!Traincar-1) (?SITVAL . #!SitVal-508)))
Old domain values of #!PVAR-1446: ((#!Traincar-1 0) (#!Truck-1 0) (#!Train-1
0))
Supported values of #!PVAR-1446: (#!Traincar-1 #!Train-1)
New domain values of #!PVAR-1446: ((#!Train-1 1) (#!Traincar-1 1))

Checking hard unary vconstraint for #!PVAR-2373: #S(VCON :ID VCON-1.50
:STATUS NIL :SOLVE-P T :VDTN #!Plan-Vdtn-1.50 :VARS (#!PVAR-2373) :CFORM
(Q-PV-COMPATIBLE #!PVAR-2372 #!PVAR-2373) :CMODALITY T :QFORM
(Q-PV-COMPATIBLE #!Hazardous-Regular-Package ?QVAR-735) :QSUBS ((?QVAR-735
. #!PVAR-2373)) :SIT :INITIAL :AVARS (#!PVAR-2372 #!PVAR-2373)).

<<< Calling Parka Structure Matcher with query:
(:AND (#!ancestor ?QVAR-735 #!Vehicle) (#!pv-Compatible=
#!Hazardous-Regular-Package ?QVAR-735)) <<< PSM query 5 result is (9
bindings): (((?QVAR-735 . #!Airplane)) ((?QVAR-735
. #!Armored-Refrigerated-Regular-Traincar)) ((?QVAR-735
. #!Armored-Regular-Traincar)) ((?QVAR-735
. #!Refrigerated-Regular-Traincar)) ((?QVAR-735 . #!Regular-Traincar))
((?QVAR-735 . #!Armored-Refrigerated-Regular-Truck)) ((?QVAR-735
. #!Armored-Regular-Truck)) ((?QVAR-735 . #!Refrigerated-Regular-Truck))
((?QVAR-735 . #!Regular-Truck)))
Old domain values of #!PVAR-2373: ((#!Regular-Truck 0) (#!Regular-Traincar
0) (#!Airplane 0) (#!Train 0))

Supported values of #!PVAR-2373: (#!Regular-Truck
#!Refrigerated-Regular-Truck #!Armored-Regular-Truck
#!Armored-Refrigerated-Regular-Truck #!Regular-Traincar
#!Refrigerated-Regular-Traincar #!Armored-Regular-Traincar
#!Armored-Refrigerated-Regular-Traincar #!Airplane) New domain values of
#!PVAR-2373: ((#!Airplane 1) (#!Regular-Traincar 1) (#!Regular-Truck 1))

;; other unary constraints are checked...
;; Variables to map using forward-checking constraint
;; satisfaction are determine.

PVAR Scores: ((#!PVAR-2362 . 1) (#!PVAR-2373 . 1) (#!PVAR-2356 . 1)
(#!PVAR-2354 . 1) (#!PVAR-2367 . 2) (#!PVAR-1447 . 5) (#!PVAR-2363 . 2)
(#!PVAR-2371 . 2) (#!PVAR-2366 . 1) (#!PVAR-2353 . 2) (#!PVAR-2369 . 1)
(#!PVAR-1441 . 7) (#!PVAR-1442 . 6) (#!PVAR-1446 . 9)) PVAR Scores:
((#!PVAR-2361 . 0) (#!PVAR-2355 . 0) (#!PVAR-2378 . 0) (#!PVAR-2360 . 3)
(#!PVAR-1445 . 0) (#!PVAR-2374 . 0) (#!PVAR-1444 . 5) (#!PVAR-1443 . 1)
(#!PVAR-2359 . 1) (#!PVAR-1450 . 0))


;; domains of unmapped plan variables are determined...


;; the following are the domains found (source pvar, target value):
(#!PVAR-1441 (#!Traincar-1 2)) (#!PVAR-1442 (#!Train-1 0)) (#!PVAR-1443
(#!City1-Ts 0)) (#!PVAR-1444 (#!City4-Ts 2)) (#!PVAR-1445 (#!Rail-Route14
```

```
1)) (#!PVAR-1446 (#!Traincar-1 1)) (#!PVAR-1447 (#!Train-1 0)) (#!PVAR-1450
(#!Rail-Route42 1)) (#!PVAR-2353 (#!Train 0)) (#!PVAR-2354
(#!Regular-Traincar 1)) (#!PVAR-2355 (#!Rail-Route42 2)) (#!PVAR-2356
(#!Rail-Route 0)) (#!PVAR-2359 (#!City1-Ts 0)) (#!PVAR-2360 (#!City1-Ts 0))
(#!PVAR-2361 (#!Rail-Route14 2)) (#!PVAR-2362 (#!Rail-Route 0)) (#!PVAR-2363
(#!Regular-Traincar 0)) (#!PVAR-2366 (#!Rail-Route 0)) (#!PVAR-2367
(#!Regular-Traincar 0)) (#!PVAR-2369 (#!Rail-Route 0)) (#!PVAR-2371 (#!Train
0)) (#!PVAR-2373 (#!Regular-Traincar 1)) (#!PVAR-2374 (#!Rail-Route42 1))
(#!PVAR-2378 (#!Rail-Route14 1))

<<< Instantiating target plan for problem TOP-PROB-1070 from source plan
#!Plan-CE-Located-At-1.5...
Adding node: (GOAL-TPNODE-1337 #!Case-Make-Deliveries-1-InitSit)
Adding node: (TPNODE-1336 NIL)
Adding node: (TPLAN-1341 #!Plan-CE-Located-At-1.5)
Adding node: (TPLAN-1481 #!Plan-CE-Transported-1.1)

;; etc.  New nodes are added to the HTN...

Instantiated target plan TPLAN-1341.
Target Plan TPLAN-1341 (Source: #!Plan-CE-Located-At-1.5) Actions:
   TACT-1492 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.1.
   TACT-1489 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.2.
   TACT-1486 - Type: #!P-Collect-Fees, Roles: ((#!package= #!Pkg-1.3)),
   Source: #!P-Collect-Fees-1.1.
   TACT-1527 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.3.
   TACT-1536 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Open-Door-1.1.
   TACT-1541 - Type: #!P-Load-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Traincar-1)), Source: #!P-Load-Package-1.1.
   TACT-1544 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Close-Door-1.1.
   TACT-1557 - Type: #!P-Attach-Traincar, Roles: ((#!train= #!Train-1)
   (#!traincar= #!Traincar-1)), Source: #!P-Attach-Traincar-1.1.
   TACT-1564 - Type: #!P-Move-Vehicle, Roles: ((#!vehicle= #!Train-1)
   (#!origin= #!City1-Ts) (#!destin= #!City4-Ts)
   (#!route= #!Rail-Route14)), Source: #!P-Move-Vehicle-1.1.
   TACT-1567 - Type: #!P-Detach-Traincar, Roles: ((#!train= #!Train-1)
   (#!traincar= #!Traincar-1)), Source: #!P-Detach-Traincar-1.1.
   TACT-1578 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Open-Door-1.2.
   TACT-1581 - Type: #!P-Unload-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Traincar-1)), Source: #!P-Unload-Package-1.1.
   TACT-1584 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Close-Door-1.2.
   TACT-1587 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.4.
   TACT-1598 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.5.
   TACT-1607 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Open-Door-1.3.
   TACT-1612 - Type: #!P-Load-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Traincar-1)), Source: #!P-Load-Package-1.2.
```

```
   TACT-1615 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Close-Door-1.3.
   TACT-1628 - Type: #!P-Attach-Traincar, Roles:
   (#!train= #!Train-1) (#!traincar= #!Traincar-1)),
   Source: #!P-Attach-Traincar-1.2.
   TACT-1635 - Type: #!P-Move-Vehicle, Roles: ((#!vehicle= #!Train-1)
   (#!origin= :UNMAPPED) (#!destin= #!City2-Ts)
   (#!route= #!Rail-Route42)), Source: #!P-Move-Vehicle-1.2.
   TACT-1638 - Type: #!P-Detach-Traincar, Roles: ((#!train= #!Train-1)
   (#!traincar= #!Traincar-1)), Source: #!P-Detach-Traincar-1.2.
   TACT-1649 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
    Source: #!P-Open-Door-1.4.
   TACT-1652 - Type: #!P-Unload-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Traincar-1)), Source: #!P-Unload-Package-1.2.
   TACT-1655 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Close-Door-1.4.
   TACT-1658 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.6.
   TACT-1661 - Type: #!P-Deliver, Roles: ((#!package= #!Pkg-1.3)),
   Source: #!P-Deliver-1.1.

;; newly instantiated source plan checked for out facts...

Checking for unnecessary goals in (TPLAN-824 #!Plan-CE-Located-At-1.5)
Checking for extra goals in TOP-PROB-553
Checking for out facts in TOP-PROB-553
Checking for out facts in target plan TPLAN-824.

Checking validation [PVDTN :ID TVDTN-825 :SOURCE #!Plan-Vdtn-1.34 :TARGET
NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit) :CONSUM
(TACT-1070 #!Do-Nothing-1.4) :COND [PCE :ID TCOND-826 :SFRAME
#!Plan-Vdtn-1.34-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]].

<<< Calling Parka Structure Matcher with query:
(#!everyInstanceOf #!Pkg-1.3 #!Hazardous-Package)
<<< PSM query 26 result is (1 bindings):
(((T . T)))

*** Noting out fact #S(VCON :ID VCON-1.34 :STATUS :CHECKED :SOLVE-P NIL
:VDTN #!Plan-Vdtn-1.34 :VARS (#!PVAR-1440) :CFORM (Q-HAS-TYPE #!PVAR-1440
#!Hazardous-Package) :CMODALITY NIL :QFORM (Q-HAS-TYPE #!Pkg-1.3
#!Hazardous-Package) :QSUBS NIL :SIT NIL :AVARS (#!PVAR-1440))

;; This out fact is the result of the source package being nonhazardous
;; and the target being hazardous.  Etc...

Checking validation [PVDTN :ID TVDTN-835 :SOURCE #!Plan-Vdtn-1.72 :TARGET
NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit) :CONSUM
(TACT-1010 #!Do-Nothing-1.3) :COND [PCE :ID TCOND-836 :SFRAME
```

```
#!Plan-Vdtn-1.72-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]].

<<< Calling Parka Structure Matcher with query:
(#!everyInstanceOf #!Pkg-1.3 #!Hazardous-Package)
<<< PSM query 31 result is (1 bindings):
(((T . T)))

*** Noting out fact
#S(VCON :ID VCON-1.72 :STATUS :CHECKED :SOLVE-P NIL :VDTN #!Plan-Vdtn-1.72
                            :VARS (#!PVAR-1440) :CFORM (Q-HAS-TYPE
                            #!PVAR-1440 #!Hazardous-Package) :CMODALITY NIL
                            :QFORM (Q-HAS-TYPE #!Pkg-1.3
                            #!Hazardous-Package) :QSUBS NIL :SIT NIL :AVARS
                            (#!PVAR-1440))

;; This out fact is the result of the source package being nonhazardous
;; and the target being hazardous.  Etc...

Checking validation [PVDTN :ID TVDTN-847 :SOURCE #!Plan-Vdtn-1.17 :TARGET
NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit) :CONSUM
(TPLAN-981 #!Plan-CE-Carried-Direct-1.1) :COND [PCE :ID TCOND-848 :SFRAME
#!Plan-Vdtn-1.17-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]].

<<< Calling Parka Structure Matcher with query:
(#!everyInstanceOf #!Pkg-1.3 #!Hazardous-Package)
<<< PSM query 37 result is (1 bindings):
(((T . T)))

*** Noting out fact
#S(VCON :ID VCON-1.17 :STATUS :CHECKED :SOLVE-P NIL :VDTN #!Plan-Vdtn-1.17
                            :VARS (#!PVAR-1440) :CFORM (Q-HAS-TYPE
                            #!PVAR-1440 #!Hazardous-Package) :CMODALITY NIL
                            :QFORM (Q-HAS-TYPE #!Pkg-1.3
                            #!Hazardous-Package) :QSUBS NIL :SIT NIL :AVARS
                            (#!PVAR-1440))

;; This out fact is the result of the source package being nonhazardous
;; and the target being hazardous.  Etc...

Checking validation [PVDTN :ID TVDTN-945 :SOURCE #!Plan-Vdtn-1.68 :TARGET
NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit) :CONSUM
(TACT-1141 #!Do-Nothing-1.6) :COND [PCE :ID TCOND-946 :SFRAME
#!Plan-Vdtn-1.68-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
```

```
#!Hazardous-Package))]].

<<< Calling Parka Structure Matcher with query:
(#!everyInstanceOf #!Pkg-1.3 #!Hazardous-Package)
<<< PSM query 86 result is (1 bindings):
(((T . T)))
*** Noting out fact #S(VCON :ID VCON-1.68 :STATUS :CHECKED :SOLVE-P NIL
                            :VDTN #!Plan-Vdtn-1.68 :VARS (#!PVAR-1440)
                            :CFORM (Q-HAS-TYPE #!PVAR-1440
                            #!Hazardous-Package) :CMODALITY NIL :QFORM
                            (Q-HAS-TYPE #!Pkg-1.3 #!Hazardous-Package)
                            :QSUBS NIL :SIT NIL :AVARS (#!PVAR-1440))

;; This out fact is the result of the source package being nonhazardous
;; and the target being hazardous.  Etc...

Checking validation [PVDTN :ID TVDTN-949 :SOURCE #!Plan-Vdtn-1.52 :TARGET
NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit) :CONSUM
(TPLAN-981 #!Plan-CE-Carried-Direct-1.1) :COND [PCE :ID TCOND-950 :SFRAME
#!Plan-Vdtn-1.52-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Available :MODAL T :ROLES ((#!thing= [TPVAR :NAME "PVAR-2361.3" :TVAL
#!Rail-Rout Checking validation [PVDTN :ID TVDTN-953 :SOURCE
#!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-818
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1081 #!Do-Nothing-1.5) :COND
[PCE :ID TCOND-954 :SFRAME #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]].

<<< Calling Parka Structure Matcher with query:
(#!everyInstanceOf #!Pkg-1.3 #!Hazardous-Package)
<<< PSM query 90 result is (1 bindings):
(((T . T)))
*** Noting out fact #S(VCON :ID VCON-1.25 :STATUS :CHECKED :SOLVE-P NIL
                            :VDTN #!Plan-Vdtn-1.25 :VARS (#!PVAR-1440)
                            :CFORM (Q-HAS-TYPE #!PVAR-1440
                            #!Hazardous-Package) :CMODALITY NIL :QFORM
                            (Q-HAS-TYPE #!Pkg-1.3 #!Hazardous-Package)
                            :QSUBS NIL :SIT NIL :AVARS (#!PVAR-1440))

;; This out fact is the result of the source package being nonhazardous
;; and the target being hazardous.  Etc...

Found out fact in validation [PVDTN :ID TVDTN-953 :SOURCE #!Plan-Vdtn-1.25
:TARGET NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit)
:CONSUM (TACT-1081 #!Do-Nothing-1.5) :COND [PCE :ID TCOND-954 :SFRAME
#!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]].
```

```
REPAIR-FAILING-FILTER-CONDITION-VALIDATION [PVDTN :ID TVDTN-953 :SOURCE
#!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-818
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1081 #!Do-Nothing-1.5) :COND
[PCE :ID TCOND-954 :SFRAME #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]]

Removing validation [PVDTN :ID TVDTN-953 :SOURCE #!Plan-Vdtn-1.25 :TARGET
NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit) :CONSUM
(TACT-1081 #!Do-Nothing-1.5) :COND [PCE :ID TCOND-954 :SFRAME
#!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]]

Adding plan refit task [PMTASK PMTASK-1146 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-553
:PLAN (TPLAN-824 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-1080
#!Plan-CE-Package-Loaded-Haz-1.2) :VDTN [PVDTN :ID TVDTN-953 :SOURCE
#!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-818
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1081 #!Do-Nothing-1.5) :COND
[PCE :ID TCOND-954 :SFRAME #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL]

Found out fact in validation [PVDTN :ID TVDTN-945 :SOURCE #!Plan-Vdtn-1.68
:TARGET NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit)
:CONSUM (TACT-1141 #!Do-Nothing-1.6) :COND [PCE :ID TCOND-946 :SFRAME
#!Plan-Vdtn-1.68-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]].

REPAIR-FAILING-FILTER-CONDITION-VALIDATION [PVDTN :ID TVDTN-945 :SOURCE
#!Plan-Vdtn-1.68 :TARGET NIL :CONTRIB (INIT-TPNODE-818
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1141 #!Do-Nothing-1.6) :COND
[PCE :ID TCOND-946 :SFRAME #!Plan-Vdtn-1.68-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]]

Removing validation [PVDTN :ID TVDTN-945 :SOURCE #!Plan-Vdtn-1.68 :TARGET
NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit) :CONSUM
(TACT-1141 #!Do-Nothing-1.6) :COND [PCE :ID TCOND-946 :SFRAME
#!Plan-Vdtn-1.68-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]]
Replacing nodes ((TACT-1141 #!Do-Nothing-1.6)) with (TPLAN-1140
```

```
#!Plan-Not-CE-Package-Loaded-Haz-1.2) in partial ordering.
Remove old reduction (splan) of node (TPLAN-1140
#!Plan-Not-CE-Package-Loaded-Haz-1.2).  Deleting node: (TACT-1141
#!Do-Nothing-1.6)

Adding plan refit task [PMTASK PMTASK-1147 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-553
:PLAN (TPLAN-824 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-1140
#!Plan-Not-CE-Package-Loaded-Haz-1.2) :VDTN [PVDTN :ID TVDTN-945 :SOURCE
#!Plan-Vdtn-1.68 :TARGET NIL :CONTRIB (INIT-TPNODE-818
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1141 #!Do-Nothing-1.6) :COND
[PCE :ID TCOND-946 :SFRAME #!Plan-Vdtn-1.68-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL]

Found out fact in validation [PVDTN :ID TVDTN-847 :SOURCE #!Plan-Vdtn-1.17
:TARGET NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit)
:CONSUM (TPLAN-981 #!Plan-CE-Carried-Direct-1.1) :COND [PCE :ID TCOND-848
:SFRAME #!Plan-Vdtn-1.17-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]].

REPAIR-FAILING-FILTER-CONDITION-VALIDATION [PVDTN :ID TVDTN-847 :SOURCE
#!Plan-Vdtn-1.17 :TARGET NIL :CONTRIB (INIT-TPNODE-818
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TPLAN-981
#!Plan-CE-Carried-Direct-1.1) :COND [PCE :ID TCOND-848 :SFRAME
#!Plan-Vdtn-1.17-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]]

Removing validation [PVDTN :ID TVDTN-847 :SOURCE #!Plan-Vdtn-1.17 :TARGET
NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit) :CONSUM
(TPLAN-981 #!Plan-CE-Carried-Direct-1.1) :COND [PCE :ID TCOND-848 :SFRAME
#!Plan-Vdtn-1.17-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]]

Adding plan refit task [PMTASK PMTASK-1149 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-553
:PLAN (TPLAN-824 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-979
#!Plan-CE-Carried-Via-Hub-1.1) :VDTN [PVDTN :ID TVDTN-847 :SOURCE
#!Plan-Vdtn-1.17 :TARGET NIL :CONTRIB (INIT-TPNODE-818
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TPLAN-981
#!Plan-CE-Carried-Direct-1.1) :COND [PCE :ID TCOND-848 :SFRAME
#!Plan-Vdtn-1.17-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
```

```
#!Hazardous-Package))]] :COND NIL :PMODS NIL]

Found out fact in validation [PVDTN :ID TVDTN-835 :SOURCE #!Plan-Vdtn-1.72
:TARGET NIL :CONTRIB (INIT-TPNODE-818 #!Case-Make-Deliveries-1-InitSit)
:CONSUM (TACT-1010 #!Do-Nothing-1.3) :COND [PCE :ID TCOND-836 :SFRAME
#!Plan-Vdtn-1.72-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]].

REPAIR-FAILING-FILTER-CONDITION-VALIDATION [PVDTN :ID TVDTN-835 :SOURCE
#!Plan-Vdtn-1.72 :TARGET NIL :CONTRIB (INIT-TPNODE-818
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-1010 #!Do-Nothing-1.3) :COND
[PCE :ID TCOND-836 :SFRAME #!Plan-Vdtn-1.72-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]]

 Calling (REPAIR-FAILING-FILTER-CONDITION-VALIDATION [PVDTN :ID TVDTN-2313
           :SOURCE #!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
           #!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2441
           #!Do-Nothing-1.5) :COND [PCE :ID TCOND-2314 :SFRAME
           #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE #!Filter-Condition
           :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR :NAME
           "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1
           :CONS NIL]) (#!type= #!Hazardous-Package))]] #<UMT-PDESC
           #xE6A061>)

REPAIR-FAILING-FILTER-CONDITION-VALIDATION [PVDTN :ID TVDTN-2313 :SOURCE
#!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2441 #!Do-Nothing-1.5) :COND
[PCE :ID TCOND-2314 :SFRAME #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]]

Adding plan refit task [PMTASK PMTASK-2514 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-1913
:PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-2440
#!Plan-CE-Package-Loaded-Haz-1.2) :VDTN [PVDTN :ID TVDTN-2313 :SOURCE
#!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2441 #!Do-Nothing-1.5) :COND
[PCE :ID TCOND-2314 :SFRAME #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL]

REPAIR-FAILING-FILTER-CONDITION-VALIDATION returned [PMTASK PMTASK-2514
:PTYPE :REREDUCE :STYPE :REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL
:PRTY 0 :PROB TOP-PROB-1913 :PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5)
:NODE (TPLAN-2440 #!Plan-CE-Package-Loaded-Haz-1.2) :VDTN [PVDTN :ID
```

```
TVDTN-2313 :SOURCE #!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2441 #!Do-Nothing-1.5) :COND
[PCE :ID TCOND-2314 :SFRAME #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL]

Selecting and Processing PMTASK [PMTASK PMTASK-2514 :PTYPE :REREDUCE :STYPE
 :REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-1913
 :PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-2440
 #!Plan-CE-Package-Loaded-Haz-1.2) :VDTN [PVDTN :ID TVDTN-2313 :SOURCE
 #!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
 #!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2441 #!Do-Nothing-1.5)
 :COND [PCE :ID TCOND-2314 :SFRAME #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE
 #!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
 :NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
 NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL]

Possible PMODs for PMTASK [PMTASK PMTASK-2514 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-1913
:PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-2440
#!Plan-CE-Package-Loaded-Haz-1.2) :VDTN [PVDTN :ID TVDTN-2313 :SOURCE
#!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2441 #!Do-Nothing-1.5) :COND
[PCE :ID TCOND-2314 :SFRAME #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL]: (#S(PMOD :ID
PMOD-2515 :STATUS :UNKNOWN :TYPE :REREDUCE-RETRIEVE :FN
PMOD-REREDUCE-RETRIEVE))

Selecting and Processing PMOD #S(PMOD :ID PMOD-2515 :STATUS :UNKNOWN :TYPE
:REREDUCE-RETRIEVE :FN PMOD-REREDUCE-RETRIEVE) Goals of #<TPPLAN #xE8E539>:
([PCE :ID TCOND-2439 :SFRAME #!Plan-CE-Package-Loaded-Haz-1.2-Goal-1 :TFRAME
NIL :GTYPE #!Precondition :STYPE #!CE-Package-Loaded-Haz :MODAL T :ROLES
((#!package= [TPVAR :NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440
:SVAL #!Pkg-1.1 :CONS NIL]) (#!vehicle= [TPVAR :NAME "PVAR-1446.3" :TVAL
#!Traincar-1 :SVAR #!PVAR-1446 :SVAL #!Traincar-1 :CONS NIL]) (#!location=
[TPVAR :NAME "PVAR-1444.3" :TVAL #!City4-Ts :SVAR #!PVAR-1444 :SVAL
#!City3-Ts :CONS NIL]))])

E-conditions of #<TPPLAN #xE8E539>: NIL

Creating subproblem for Task [PMTASK PMTASK-2514 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-1913
:PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-2440
#!Plan-CE-Package-Loaded-Haz-1.2) :VDTN [PVDTN :ID TVDTN-2313 :SOURCE
#!Plan-Vdtn-1.25 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2441 #!Do-Nothing-1.5) :COND
[PCE :ID TCOND-2314 :SFRAME #!Plan-Vdtn-1.25-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
```

```
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS #S(PMOD :ID
PMOD-2515 :STATUS :UNKNOWN :TYPE :REREDUCE-RETRIEVE :FN
PMOD-REREDUCE-RETRIEVE)], PMOD #S(PMOD :ID PMOD-2515 :STATUS :UNKNOWN :TYPE
:REREDUCE-RETRIEVE :FN PMOD-REREDUCE-RETRIEVE) Target Problem PROB-2524
(NIL) (Class UMT-PDESC): Domain: :UM-TRANSLOG ("UM-Translog Domain") Case:
NIL, Plan: NIL, Prev. Plans: NIL RPLANs: (RPLAN-2525) Source PPLANS: NIL
Target PPLANS: NIL Root PPLAN: NIL (NIL) Subproblems: NIL Features:
(UMT-PLAN-GOAL-FEAT-2520) Mappings: NIL Rtvl. Plan RPLAN-2525 (RPLAN)
Derived-via: :AUTO Problem: PROB-2524 Features: (UMT-PLAN-GOAL-FEAT-2520)
RDESCs: (RDESC-2526) Status: NIL Succ. RDESC: NIL Sources Rtvd: NIL
Srcs. Chosen: NIL <<< Retrieving plan using RPLAN RPLAN-2525...  Rtvl. Plan
RPLAN-2525 (RPLAN) Derived-via: :AUTO Problem: PROB-2524 Features:
(UMT-PLAN-GOAL-FEAT-2520) RDESCs: (RDESC-2526) Status: NIL Succ. RDESC: NIL
Sources Rtvd: NIL Srcs. Chosen: NIL Rtvl. Description RDESC-2526
Derived-from: NIL Derived-via: :AUTO Rtvl. Plan: RPLAN-2525 Query Forms: NIL
Query Vars.: NIL Query Source: NIL Query Status: NIL Query Result: NIL
Features: (UMT-PLAN-GOAL-FEAT-2520) Mappings: NIL

<<< Beginning RDESC search from initial rdescs: (RDESC-2526)
      Parameters: Query Mode       (*rds-query-mode*): T
                  Node Succs Fn (*rds-node-succs-fn*): NIL
                  Node Score Fn (*rds-node-score-fn*):
                  #<Compiled-function GET-RDSN-BASE-SCORE #xC8D5C6>
                  Order Open Fn (*rds-order-open-fn*):
                  #<Compiled-function RDS-ORDER-OPEN-BEST #xC8EC86>
                  Eval Match Fn (*rds-eval-match-fn*):
                  #<Compiled-function NEW-PLANS-FOUND #xC90746>
                  Ignore Seen States? (*rds-ignore-seen-states-p*): T

Current OPEN list:
((NODE-2527 RDESC-2526 NIL 0 NIL))

<<< Iteration 1:

<<< Expanding New NODE-2527: Old RDESC-2526 (node NIL), New RDESC-2526

<<< Calling Parka Structure Matcher with query:
(:AND (#!goal-of-Plan ?GOAL ?PLAN) (#!instanceOf ?PKG-PVAR #!Plan-Variable)
 (#!var-Value= ?PKG-PVAR ?PKG) (#!package= ?GOAL ?PKG-PVAR) (#!instanceOf
 ?PLAN #!E-Plan) (#!modality= ?GOAL #!True) (#!instanceOf ?GOAL
 #!CE-Package-Loaded-Haz) (#!everyInstanceOf ?PKG
 #!Hazardous-Regular-Package))

<<< PSM query 95 result is (1 bindings):
(((?GOAL . #!Plan-CE-Package-Loaded-Haz-2.1-Goal-1) (?PLAN
  . #!Plan-CE-Package-Loaded-Haz-2.1) (?PKG . #!Pkg-1.2) (?PKG-PVAR
  . #!PVAR-1358)))
<<< New query forms were: NIL

<<< Done with RDESC Search.  1 nodes expanded Search Status is
```

```
    SUCCESS.
Found goal node [NODE-2527: (status 1) (rdesc-old RDESC-2526)
    (op NIL) (rdesc-new RDESC-2526) (score NIL) (cost 0) (parent NIL)
    (children NIL)] RDESC RDESC-2526: Query Status :SUCCESS Feature
    UMT-PLAN-GOAL-FEAT-2520 (orig UMT-PLAN-GOAL-FEAT-2520): Facet Bindings
    ((PLAN NIL) (GOAL TCOND-2439) (PKG #!Pkg-1.3))
Final OPEN list: NIL

Final CLOSED list: ((NODE-2527 RDESC-2526 NIL 0 NIL))

<<< Retrieved 1 candidate source plans:
   (#!Plan-CE-Package-Loaded-Haz-2.1)

<<< Selecting high scoring plan "Plan-CE-Package-Loaded-Haz-2.1" (SDESC
         SPLAN-2533), score = NIL: Features: (UMT-PLAN-GOAL-FEAT-2528)
         Mappings: ((UMT-PLAN-GOAL-FEAT-2520 . UMT-PLAN-GOAL-FEAT-2528))

Actions of source plan #!Plan-CE-Package-Loaded-Haz-2.1:
(#!P-Affix-Warning-Signs-2.1)

<<< Mapping source plan #!Plan-CE-Package-Loaded-Haz-2.1 to problem
PROB-2524...

;; etc...
;; Newly instantiated target plan is merged into overall target plan.
;; Action for attaching warning signs before loading replaces a
;; null (do-nothing) action.

Target Plan TPLAN-2184 (Source: #!Plan-CE-Located-At-1.5) Actions:
   TACT-2335 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.1.
   TACT-2332 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.2.
   TACT-2329 - Type: #!P-Collect-Fees, Roles: ((#!package= #!Pkg-1.3)),
   Source: #!P-Collect-Fees-1.1.
   TACT-2370 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.3.
   TACT-2379 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Open-Door-1.1.
   TACT-2384 - Type: #!P-Load-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Traincar-1)), Source: #!P-Load-Package-1.1.
   TACT-2387 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Close-Door-1.1.
   TACT-2400 - Type: #!P-Attach-Traincar, Roles: ((#!train= #!Train-1)
   (#!traincar= #!Traincar-1)), Source: #!P-Attach-Traincar-1.1.
   TACT-2407 - Type: #!P-Move-Vehicle, Roles: ((#!vehicle= #!Train-1)
   (#!origin= #!City1-Ts) (#!destin= #!City4-Ts)
  (#!route= #!Rail-Route14)), Source: #!P-Move-Vehicle-1.1.
   TACT-2410 - Type: #!P-Detach-Traincar, Roles: ((#!train= #!Train-1)
   (#!traincar= #!Traincar-1)), Source: #!P-Detach-Traincar-1.1.
   TACT-2421 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Open-Door-1.2.
   TACT-2424 - Type: #!P-Unload-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Traincar-1)), Source: #!P-Unload-Package-1.1.
```

```
   TACT-2427 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Close-Door-1.2.
   TACT-2430 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.4.
   TACT-2556 - Type: #!P-Affix-Warning-Signs, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= :UNMAPPED)), Source: #!P-Affix-Warning-Signs-2.1.
   TACT-2450 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Open-Door-1.3.
   TACT-2455 - Type: #!P-Load-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Traincar-1)), Source: #!P-Load-Package-1.2.
   TACT-2458 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Close-Door-1.3.
   TACT-2471 - Type: #!P-Attach-Traincar, Roles: ((#!train= #!Train-1)
   (#!traincar= #!Traincar-1)), Source: #!P-Attach-Traincar-1.2.
   TACT-2478 - Type: #!P-Move-Vehicle, Roles: ((#!vehicle= #!Train-1)
   (#!origin= :UNMAPPED) (#!destin= #!City2-Ts)
   (#!route= #!Rail-Route42)), Source: #!P-Move-Vehicle-1.2.
   TACT-2481 - Type: #!P-Detach-Traincar, Roles: ((#!train= #!Train-1)
   (#!traincar= #!Traincar-1)), Source: #!P-Detach-Traincar-1.2.
   TACT-2492 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Open-Door-1.4.
   TACT-2495 - Type: #!P-Unload-Package, Roles: ((#!package= #!Pkg-1.3)
   (#!vehicle= #!Traincar-1)), Source: #!P-Unload-Package-1.2.
   TACT-2498 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
   Source: #!P-Close-Door-1.4.
   TACT-2501 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.6.
   TACT-2504 - Type: #!P-Deliver, Roles: ((#!package= #!Pkg-1.3)),
   Source: #!P-Deliver-1.1.

;; Similarly, the other load subplans and two unload actions are replaced
;; by newly instantiated subplans for hazardous packages...

;; The old null get permit subplan is replaced by a real subplan for
;; obtaining a permit for the hazardous package...

[PVDTN :ID TVDTN-2193 :SOURCE #!Plan-Vdtn-1.62 :TARGET NIL :CONTRIB
(INIT-TPNODE-2178 #!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2335
#!Do-Nothing-1.1) :COND [PCE :ID TCOND-2194 :SFRAME #!Plan-Vdtn-1.62-Cond
:TFRAME NIL :GTYPE #!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES
((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440
:SVAL #!Pkg-1.1 :CONS NIL]) (#!type= #!Hazardous-Package))]]

Replacing nodes ((TACT-2335 #!Do-Nothing-1.1)) with (TPLAN-2334
#!Plan-CE-Have-Pkg-Permit-1.1) in partial ordering.
Remove old reduction (splan) of node (TPLAN-2334
#!Plan-CE-Have-Pkg-Permit-1.1).
Deleting node: (TACT-2335 #!Do-Nothing-1.1)

Adding plan refit task [PMTASK PMTASK-2566 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-1913
:PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-2334
#!Plan-CE-Have-Pkg-Permit-1.1) :VDTN [PVDTN :ID TVDTN-2193 :SOURCE
```

```
#!Plan-Vdtn-1.62 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2335 #!Do-Nothing-1.1) :COND
[PCE :ID TCOND-2194 :SFRAME #!Plan-Vdtn-1.62-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL]

REPAIR-FAILING-FILTER-CONDITION-VALIDATION returned [PMTASK PMTASK-2566
:PTYPE :REREDUCE :STYPE :REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL
:PRTY 0 :PROB TOP-PROB-1913 :PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5)
:NODE (TPLAN-2334 #!Plan-CE-Have-Pkg-Permit-1.1) :VDTN [PVDTN :ID TVDTN-2193
:SOURCE #!Plan-Vdtn-1.62 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2335 #!Do-Nothing-1.1) :COND
[PCE :ID TCOND-2194 :SFRAME #!Plan-Vdtn-1.62-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL] [PMTASK
PMTASK-2566 :PTYPE :REREDUCE :STYPE :REPLACE-REDUCTION :STATUS :PENDING
:PARENT NIL :PRTY 0 :PROB TOP-PROB-1913 :PLAN (TPLAN-2184
#!Plan-CE-Located-At-1.5) :NODE (TPLAN-2334 #!Plan-CE-Have-Pkg-Permit-1.1)
:VDTN [PVDTN :ID TVDTN-2193 :SOURCE #!Plan-Vdtn-1.62 :TARGET NIL :CONTRIB
(INIT-TPNODE-2178 #!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2335
#!Do-Nothing-1.1) :COND [PCE :ID TCOND-2194 :SFRAME #!Plan-Vdtn-1.62-Cond
:TFRAME NIL :GTYPE #!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES
((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440
:SVAL #!Pkg-1.1 :CONS NIL]) (#!type= #!Hazardous-Package))]] :COND NIL
:PMODS NIL] ? (modify-plan *target-pdesc*)

Selecting and Processing PMTASK [PMTASK PMTASK-2566 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-1913
:PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-2334
#!Plan-CE-Have-Pkg-Permit-1.1) :VDTN [PVDTN :ID TVDTN-2193 :SOURCE
#!Plan-Vdtn-1.62 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2335 #!Do-Nothing-1.1) :COND
[PCE :ID TCOND-2194 :SFRAME #!Plan-Vdtn-1.62-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL]

Possible PMODs for PMTASK [PMTASK PMTASK-2566 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-1913
:PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-2334
#!Plan-CE-Have-Pkg-Permit-1.1) :VDTN [PVDTN :ID TVDTN-2193 :SOURCE
#!Plan-Vdtn-1.62 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2335 #!Do-Nothing-1.1) :COND
[PCE :ID TCOND-2194 :SFRAME #!Plan-Vdtn-1.62-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS NIL]: (#S(PMOD :ID
PMOD-2567 :STATUS :UNKNOWN :TYPE :REREDUCE-RETRIEVE :FN
PMOD-REREDUCE-RETRIEVE))
```

```
Selecting and Processing PMOD #S(PMOD :ID PMOD-2567 :STATUS :UNKNOWN :TYPE
:REREDUCE-RETRIEVE :FN PMOD-REREDUCE-RETRIEVE) Goals of #<TPPLAN #xE8A299>:
([PCE :ID TCOND-2333 :SFRAME #!Plan-CE-Have-Pkg-Permit-1.1-Goal-1 :TFRAME
NIL :GTYPE #!Precondition :STYPE #!CE-Have-Pkg-Permit :MODAL T :ROLES
((#!package= [TPVAR :NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440
:SVAL #!Pkg-1.1 :CONS NIL]))])

E-conditions of #<TPPLAN #xE8A299>: NIL

Creating subproblem for Task [PMTASK PMTASK-2566 :PTYPE :REREDUCE :STYPE
:REPLACE-REDUCTION :STATUS :PENDING :PARENT NIL :PRTY 0 :PROB TOP-PROB-1913
:PLAN (TPLAN-2184 #!Plan-CE-Located-At-1.5) :NODE (TPLAN-2334
#!Plan-CE-Have-Pkg-Permit-1.1) :VDTN [PVDTN :ID TVDTN-2193 :SOURCE
#!Plan-Vdtn-1.62 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2335 #!Do-Nothing-1.1) :COND
[PCE :ID TCOND-2194 :SFRAME #!Plan-Vdtn-1.62-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND NIL :PMODS #S(PMOD :ID
PMOD-2567 :STATUS :UNKNOWN :TYPE :REREDUCE-RETRIEVE :FN
PMOD-REREDUCE-RETRIEVE)], PMOD #S(PMOD :ID PMOD-2567 :STATUS :UNKNOWN :TYPE
:REREDUCE-RETRIEVE :FN PMOD-REREDUCE-RETRIEVE) Target Problem PROB-2576
(NIL) (Class UMT-PDESC): Domain: :UM-TRANSLOG ("UM-Translog Domain") Case:
NIL, Plan: NIL, Prev. Plans: NIL RPLANs: (RPLAN-2577) Source PPLANS: NIL
Target PPLANS: NIL Root PPLAN: NIL (NIL) Subproblems: NIL Features:
(UMT-PLAN-GOAL-FEAT-2572) Mappings: NIL Rtvl. Plan RPLAN-2577 (RPLAN)
Derived-via: :AUTO Problem: PROB-2576 Features: (UMT-PLAN-GOAL-FEAT-2572)
RDESCs: (RDESC-2578) Status: NIL Succ. RDESC: NIL Sources Rtvd: NIL
Srcs. Chosen: NIL

<<< Retrieving plan using RPLAN RPLAN-2577...
Rtvl. Plan RPLAN-2577 (RPLAN) Derived-via: :AUTO Problem: PROB-2576
Features: (UMT-PLAN-GOAL-FEAT-2572) RDESCs: (RDESC-2578) Status: NIL
Succ. RDESC: NIL Sources Rtvd: NIL Srcs. Chosen: NIL Rtvl. Description
RDESC-2578 Derived-from: NIL Derived-via: :AUTO Rtvl. Plan: RPLAN-2577 Query
Forms: NIL Query Vars.: NIL Query Source: NIL Query Status: NIL Query
Result: NIL Features: (UMT-PLAN-GOAL-FEAT-2572) Mappings: NIL

<<< Beginning RDESC search from initial rdescs: (RDESC-2578)
      Parameters: Query Mode       (*rds-query-mode*): T
                  Node Succs Fn (*rds-node-succs-fn*): NIL
                  Node Score Fn (*rds-node-score-fn*):
                  #<Compiled-function GET-RDSN-BASE-SCORE #xC8D5C6>
                  Order Open Fn (*rds-order-open-fn*):
                  #<Compiled-function RDS-ORDER-OPEN-BEST #xC8EC86>
                  Eval Match Fn (*rds-eval-match-fn*):
                  #<Compiled-function NEW-PLANS-FOUND #xC90746>
                  Ignore Seen States? (*rds-ignore-seen-states-p*): T

Current OPEN list:
```

```
((NODE-2579 RDESC-2578 NIL 0 NIL))

<<< Iteration 1:

<<< Expanding New NODE-2579: Old RDESC-2578 (node NIL), New RDESC-2578

<<< Calling Parka Structure Matcher with query:
(:AND (#!goal-of-Plan ?GOAL ?PLAN) (#!instanceOf ?PKG-PVAR #!Plan-Variable)
 (#!var-Value= ?PKG-PVAR ?PKG) (#!package= ?GOAL ?PKG-PVAR) (#!instanceOf
 ?PLAN #!E-Plan) (#!modality= ?GOAL #!True) (#!instanceOf ?GOAL
 #!CE-Have-Pkg-Permit) (#!everyInstanceOf ?PKG #!Hazardous-Regular-Package))
<<< PSM query 97 result is (2 bindings):
(((?GOAL . #!Plan-CE-Have-Pkg-Permit-2.2-Goal-1) (?PLAN
  . #!Plan-CE-Have-Pkg-Permit-2.2) (?PKG . #!Pkg-1.2) (?PKG-PVAR
  . #!PVAR-1358)) ((?GOAL . #!Plan-CE-Have-Pkg-Permit-2.1-Goal-1) (?PLAN
  . #!Plan-CE-Have-Pkg-Permit-2.1) (?PKG . #!Pkg-1.2) (?PKG-PVAR
  . #!PVAR-1358)))
<<< New query forms were: NIL

<<< Done with RDESC Search.  1 nodes expanded Search Status is
    SUCCESS.
Found goal node [NODE-2579: (status 1) (rdesc-old RDESC-2578)
    (op NIL) (rdesc-new RDESC-2578) (score NIL) (cost 0) (parent NIL)
    (children NIL)] RDESC RDESC-2578: Query Status :SUCCESS Feature
    UMT-PLAN-GOAL-FEAT-2572 (orig UMT-PLAN-GOAL-FEAT-2572): Facet Bindings
    ((PLAN NIL) (GOAL TCOND-2333) (PKG #!Pkg-1.3))

Final OPEN list:
NIL

Final CLOSED list:
((NODE-2579 RDESC-2578 NIL 0 NIL))

<<< Retrieved 2 candidate source plans: (#!Plan-CE-Have-Pkg-Permit-2.2
   #!Plan-CE-Have-Pkg-Permit-2.1) Ignoring Plan
   #!Plan-CE-Have-Pkg-Permit-2.2, a superplan of
   #!Plan-CE-Have-Pkg-Permit-2.1.

<<< Selecting high scoring plan "Plan-CE-Have-Pkg-Permit-2.1" (SDESC
         SPLAN-2590), score = NIL: Features: (UMT-PLAN-GOAL-FEAT-2585)
         Mappings: ((UMT-PLAN-GOAL-FEAT-2572 . UMT-PLAN-GOAL-FEAT-2585))
Actions of source plan #!Plan-CE-Have-Pkg-Permit-2.1: (#!P-Obtain-Permit-2.1)

<<< Mapping source plan #!Plan-CE-Have-Pkg-Permit-2.1 to problem PROB-2576...
PVAR Scores: NIL
PVAR Scores: NIL

Variables to bind (in order): NIL

Vars without active constraints: NIL
GET-PVAR-DOMAINS: pvars=NIL, dset=NIL
```

```
Active Constraints:
NIL
Done with MAP-PLAN

<<< Instantiating target plan for problem PROB-2576 from source plan
#!Plan-CE-Have-Pkg-Permit-2.1...

Instantiated target plan TPLAN-2601.

Target Plan TPLAN-2601 (Source: #!Plan-CE-Have-Pkg-Permit-2.1) Actions:
TACT-2604 - Type: #!P-Obtain-Permit, Roles: ((#!package= #!Pkg-1.3)),
Source: #!P-Obtain-Permit-2.1.

<<< Processing replacement plan (TPLAN-2601 #!Plan-CE-Have-Pkg-Permit-2.1)
for problem PROB-2576...
<<< Checking replacement plan (TPLAN-2601
#!Plan-CE-Have-Pkg-Permit-2.1) for problem PROB-2576...

Checking E-Preconditions of (TPLAN-2601 #!Plan-CE-Have-Pkg-Permit-2.1)
Using contributor from [PVDTN :ID TVDTN-2548 :SOURCE #!Plan-Vdtn-2.22
:TARGET NIL :CONTRIB (INIT-TPNODE-2178 #!Case-Make-Deliveries-1-InitSit)
:CONSUM (TACT-2556 #!P-Affix-Warning-Signs-2.1) :COND [PCE :ID TCOND-2549
:SFRAME #!Plan-Vdtn-2.22-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL T :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]] for [PVDTN :ID TVDTN-2602 :SOURCE #!Plan-Vdtn-2.26
:TARGET NIL :CONTRIB (INIT-TPNODE-2178 #!Case-Make-Deliveries-1-InitSit)
:CONSUM (TACT-2604 #!P-Obtain-Permit-2.1) :COND [PCE :ID TCOND-2603 :SFRAME
#!Plan-Vdtn-2.26-Cond :TFRAME NIL :GTYPE #!Filter-Condition :STYPE
#!CE-Has-Type :MODAL T :ROLES ((#!thing= [TPVAR :NAME "PVAR-1440.3" :TVAL
#!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS NIL]) (#!type=
#!Hazardous-Package))]] in #<TPPLAN #xF30529>.

Checking E-Conditions of (TPLAN-2334 #!Plan-CE-Have-Pkg-Permit-1.1)

Adding plan refit task [PMTASK PMTASK-2618 :PTYPE :PRUNE :STYPE :PLAN-RR
:STATUS :PENDING :PARENT PMTASK-2566 :PRTY 3 :PROB TOP-PROB-1913 :PLAN
(TPLAN-2184 #!Plan-CE-Located-At-1.5) :NODE NIL :VDTN [PVDTN :ID TVDTN-2193
:SOURCE #!Plan-Vdtn-1.62 :TARGET NIL :CONTRIB (INIT-TPNODE-2178
#!Case-Make-Deliveries-1-InitSit) :CONSUM (TACT-2335 #!Do-Nothing-1.1) :COND
[PCE :ID TCOND-2194 :SFRAME #!Plan-Vdtn-1.62-Cond :TFRAME NIL :GTYPE
#!Filter-Condition :STYPE #!CE-Has-Type :MODAL NIL :ROLES ((#!thing= [TPVAR
:NAME "PVAR-1440.3" :TVAL #!Pkg-1.3 :SVAR #!PVAR-1440 :SVAL #!Pkg-1.1 :CONS
NIL]) (#!type= #!Hazardous-Package))]] :COND :CONDIT :PMODS NIL]

;; unused validations are pruned...

Target Plan TPLAN-2184 (Source: #!Plan-CE-Located-At-1.5) Actions:
   TACT-2604 - Type: #!P-Obtain-Permit, Roles: ((#!package= #!Pkg-1.3)),
   Source: #!P-Obtain-Permit-2.1.
```

```
TACT-2332 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.2.
TACT-2329 - Type: #!P-Collect-Fees, Roles: ((#!package= #!Pkg-1.3)),
Source: #!P-Collect-Fees-1.1.
TACT-2370 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.3.
TACT-2379 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
Source: #!P-Open-Door-1.1.
TACT-2384 - Type: #!P-Load-Package, Roles: ((#!package= #!Pkg-1.3)
(#!vehicle= #!Traincar-1)), Source: #!P-Load-Package-1.1.
TACT-2387 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
 Source: #!P-Close-Door-1.1.
TACT-2400 - Type: #!P-Attach-Traincar, Roles: ((#!train= #!Train-1)
(#!traincar= #!Traincar-1)), Source: #!P-Attach-Traincar-1.1.
TACT-2407 - Type: #!P-Move-Vehicle, Roles: ((#!vehicle= #!Train-1)
(#!origin= #!City1-Ts) (#!destin= #!City4-Ts)
(#!route= #!Rail-Route14)), Source: #!P-Move-Vehicle-1.1.
TACT-2410 - Type: #!P-Detach-Traincar, Roles: ((#!train= #!Train-1)
(#!traincar= #!Traincar-1)), Source: #!P-Detach-Traincar-1.1.
TACT-2421 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
Source: #!P-Open-Door-1.2.
TACT-2424 - Type: #!P-Unload-Package, Roles: ((#!package= #!Pkg-1.3)
(#!vehicle= #!Traincar-1)), Source: #!P-Unload-Package-1.1.
TACT-2427 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
Source: #!P-Close-Door-1.2.
TACT-2430 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.4.
TACT-2556 - Type: #!P-Affix-Warning-Signs, Roles: ((#!package= #!Pkg-1.3)
(#!vehicle= :UNMAPPED)), Source: #!P-Affix-Warning-Signs-2.1.
TACT-2450 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
Source: #!P-Open-Door-1.3.
TACT-2455 - Type: #!P-Load-Package, Roles: ((#!package= #!Pkg-1.3)
(#!vehicle= #!Traincar-1)), Source: #!P-Load-Package-1.2.
TACT-2458 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
Source: #!P-Close-Door-1.3.
TACT-2471 - Type: #!P-Attach-Traincar, Roles: ((#!train= #!Train-1)
(#!traincar= #!Traincar-1)), Source: #!P-Attach-Traincar-1.2.
TACT-2478 - Type: #!P-Move-Vehicle, Roles: ((#!vehicle= #!Train-1)
(#!origin= :UNMAPPED) (#!destin= #!City2-Ts)
(#!route= #!Rail-Route42)), Source: #!P-Move-Vehicle-1.2.
TACT-2481 - Type: #!P-Detach-Traincar, Roles: ((#!train= #!Train-1)
(#!traincar= #!Traincar-1)), Source: #!P-Detach-Traincar-1.2.
TACT-2492 - Type: #!P-Open-Door, Roles: ((#!vehicle= #!Traincar-1)),
Source: #!P-Open-Door-1.4.
TACT-2495 - Type: #!P-Unload-Package, Roles: ((#!package= #!Pkg-1.3)
(#!vehicle= #!Traincar-1)), Source: #!P-Unload-Package-1.2.
TACT-2498 - Type: #!P-Close-Door, Roles: ((#!vehicle= #!Traincar-1)),
Source: #!P-Close-Door-1.4.
TACT-2501 - Type: #!Do-Nothing, Roles: NIL, Source: #!Do-Nothing-1.6.
TACT-2504 - Type: #!P-Deliver, Roles: ((#!package= #!Pkg-1.3)),
Source: #!P-Deliver-1.1.
```

Chapter 8

Other Connections

The CAPER research has connections to several other areas of work in artificial intelligence and cognitive science in general. This chapter discusses some of these connections, thus tying up a few loose ends. These connections have influenced the design of CAPER and its application. Section 8.1 briefly discusses this research methodology, which explains some of the design decisions taken in CAPER and may be helpful to other designers of CBP/CBR systems.

Like most CBP systems, CAPER does transformational analogy, in which a complete source plan(s) is transformed (adapted) to a target plan. Some CBP systems do derivational analogy in which cases consisting of previous planning decisions are reused during plan generation (e.g., PRODIGY/ANALOGY (Veloso, 1992)). As a new plan is being produced and decisions are being made (e.g., which goal to expand, which operator to apply, etc.), old cases involving similar decisions are consulted. Systems like PRIAR (Kambhampati and Hendler, 1992) and CAPER could be considered a kind of hybrid as they transform old plans but also make use of the hierarchical task network and plan validation structure, both of which capture derivational information of a sort. Section 8.2 presents some work that investigates the connection between CAPER and derivational analogy.

Section 8.3 discusses some work in cognitive science that provided inspiration for the design of CAPER, as well as for many previous CBR/CBP systems. The research methodology in CAPER was to apply ideas from psychology to design a system with more of the parallel and associative character of human memory retrieval but *not* to model specific human cognitive processes. Although human memory recall is far from perfect, it is highly flexible: a given set of cues may produce many associations and retrieval can involve (recursive) retrieval of additional cues (e.g., (Riesbeck and Schank, 1989; Williams and Hollan, 1981)). Much relevant work has been done on human memory and human analogical reasoning.

8.1 Research Methodology

The CAPER project grew out of two primary impetuses: the desire to apply the fast memory of PARKA to some reasoning task and the belief that case-based reasoning could be useful in solving planning problems (as CHEF (Hammond, 1986), for example, had shown).

CAPER, as the first PARKA application, showed the need for additional expressivity in the PARKA language, both in the storage of knowledge and in the formation of queries. Representing knowledge in PARKA was facilitated by the introduction (into version 3) of the distinction between categories and instances (and hence is-a and instance-of links), the first-class status of predicates, predicate inverses, superslots/subslots, multiple slot fillers, and set/vector slot fillers. (See Section 3.1.1 for a brief description of these.) CAPER also provided an impetus for the addition of the PARKA Structure Matcher (Section 3.1.4) into version 2 (and subsequent versions) of PARKA. This was prompted by the need to match structural relationships in the target cases to source cases.

The flexible representation and retrieval of complex previous cases with many relationships among their parts is a task for which CAPER is well-suited. An initial inspiration for CAPER's design came from plans composed of recurring subplans. In Mexican cooking, for example, a plan for making an enchilada is similar to a plan for making a burrito. Both involve making a wrapper (tortilla), making a filling, rolling the filling in the wrapper, etc.

If a CBP system was making a kind of Mexican food for the first time, it could retrieve subplans from similar foods and merge these subplans into a single plan. The CBP system would need to go to memory several times, at least once for each subplan, and would need to be able to retrieve subplans. Thus retrieval would need a finer granularity than that of an entire plan (case) and would need greater flexibility than that provided in systems that used preindexed casebases. The need to represent the structural relationships among subplans, actions, and objects and to match a target problem to a source plan using them became design goals eventually realized in CAPER. Additional ideas for exploiting these capabilities are discussed in the next chapter.

8.2 The Derivational Analogy Connection

As discussed in the previous chapters, the flexible case representation and retrieval mechanisms of CAPER have proven useful in doing transformational analogy. The utility of these mechanisms for doing derivational analogy has also been investigated. As mentioned above, CAPER's storage and use of validation structure and hierarchical task decomposition data, generated during planning, is similar to some of the techniques used for derivational analogy.

Several of CAPER's mechanisms were used by SEQER (Kettler and Darden, 1993), a prototype system that plans protein sequencing experiments using derivational analogy.[1] SEQER's architecture is designed to be applicable to various scientific experiment planning domains. SEQER is described in detail in (Kettler and Darden, 1993), parts of which are excerpted here to illustrate how mechanisms of CAPER can be used to support derivational analogy in a real-world domain.

The use of derivational analogy allows SEQER to reuse previous planning experiences from similar situations and to function in the absence of strong domain knowledge. As it solves new problems, SEQER accumulates additional planning experience which it encodes

---

[1] "SEQER" is pronounced as "seeker".

as new cases. These cases can be efficiently retrieved from the casebase using the fast and flexible retrieval mechanisms of CAPER/PARKA.

SEQER was initially tested in the domain of protein sequencing experiment planning, in which the objective is to determine the amino acid sequence (and other properties) of a protein. While protein sequencing can be done using sequencing machines[2], the domain of protein sequencing using manual methods was chosen as the initial testbed for SEQER because it is well understood and a simulator for the domain is available. The task of manually sequencing a protein is a complex one amenable to automated planning.

Experiment planning is a complex task. The experiment should efficiently and inexpensively achieve its goals (e.g., test a hypothesis, determine a property, etc.). The use of resources (e.g., money, time, people, equipment, supplies) should be minimized. Planning may be complicated due to the unavailability of strong domain knowledge, established methodologies, or experienced personnel. In addition, uncertainty about experimental conditions, instrument readings, and the objects being manipulated must be dealt with. The results of executing a laboratory procedure may not be completely predictable when domain knowledge is lacking or uncertainty exists. Thus, planning may need to be interleaved with execution such that the results of previous procedures help to determine the next procedure to execute.

This interleaving is necessary in experiments to sequence an unknown protein (polypeptide). The sequence of a protein (often hundreds of amino acids long), along with its shape and other properties, determines the structure (and hence the function) of the protein. There are 20^{200} possible sequences for a polypeptide of average length. SEQER plans an experiment to determine a polypeptide's amino acid sequence, N-terminal and C-terminal end groups (i.e., Formyl, Acetyl, Pyroglutamyl or Amide blocked, or unblocked), circularity, and disulfide bond locations (if any).

Various laboratory procedures can be performed to achieve these goals including acid and base hydrolysis, chemical and enzymatic cleavage, oxidation, Edman degradation, end group analysis, exopeptidase degradation, etc. The execution of many of these procedures is realistically simulated by the SequenceIt program (Place and Schmidt, 1992), a tutorial program for protein sequencing techniques which we are using to test SEQER's plans in lieu of an actual laboratory. SequenceIt provides 35 laboratory procedures to choose from, each of which has several parameters that can be varied.

SEQER's top-level processes and modules are shown in Figure 8.1. The domain-independent Planner component consists of the Experiment Controller, Rule Applier, Case Applier, and Decision Episode Storer modules (shown as dark-outlined boxes). These last two modules, in conjunction with mechanisms of CAPER, implement the reasoning by analogy. The Rule Applier module matches and fires rules. The Experiment Controller module runs the top-level process.

SEQER uses a variety of kinds of knowledge to determine the subgoals of the experiment, the tasks that achieve them, and the task to be executed next. This knowledge is contained

---

[2] The accuracy of these machines varies from 90-95 percent, depending on make, model, etc.

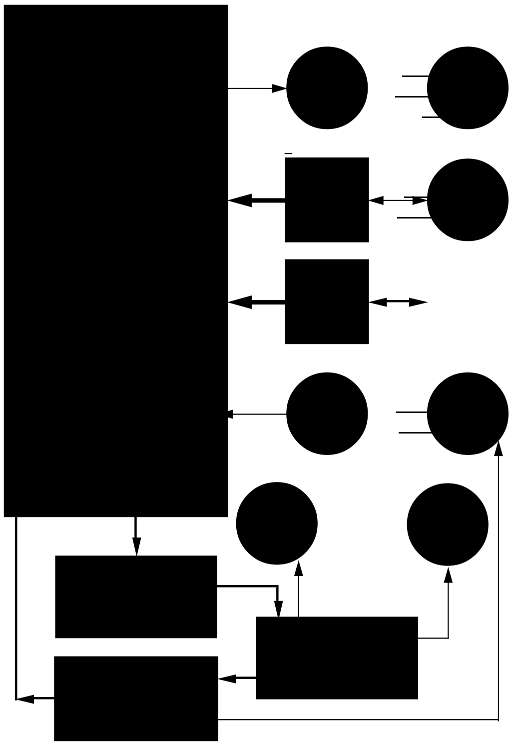

Figure 8.1: The SEQER architecture. (Boxes indicate components. Circles indicate knowledge sources. Circles with underlined labels are persistent knowledge.)

in the KB component, which contains both working knowledge[3]. Most of the KB is stored in PARKA. This knowledge includes concepts pertaining to domain objects (e.g., "Polypeptide", "Amino Acid", "Disulfide Bond"), properties (e.g., "Amino Acid Composition", "Sequence Length"), and procedures (e.g., "Acid Hydrolysis", "Clostripain Cleavage"). There are currently 450 concept frames in the protein sequencing KB. Other knowledge include the (sub)goals of the experiment, the tasks (laboratory or data analysis procedures) for achieving outstanding goals, the protein samples created, the lab state (current facts about objects, equipment, etc.), and hypotheses about the constituent amino acids and bonds of the protein being sequenced. Hypotheses are entered by the user based on his/her interpretation of the results of a lab or data procedure that was executed via SequenceIt.

Given the overall goal(s) of the experiment and information about the initial situation, SEQER produces a plan. Planning and execution are interleaved in an experiment: after each procedure is selected by SEQER, it is "executed" by the user who then interprets the results of the procedure and supplies SEQER with new hypotheses to investigate.

---

[3] A.k.a. "working memory"

For protein sequencing experiments, the input goal is to determine the amino acid sequence and other properties of a polypeptide. As initial input, SEQER is told of the quantity (and any other known properties) of a polypeptide sample whose properties are to be determined (via SequenceIt). It is also told of any unusual situations such as any lab equipment that is not functioning or other constraints (time limits, etc.) on the plan to be produced.

SEQER captures planning experience in the form of decision episodes and experiment episodes. Both of these kinds of episodes are stored as cases in the casebase. An experiment episode describes an experiment that was performed and consists of a sequence of decision episodes. A decision episode (DE) describes the choice of a task (laboratory or data procedure) to execute. A DE consists of the set of tasks chosen from, the task chosen, and the reason(s) for the choice made. DEs capture control knowledge which can be reused in analogous situations. For example, suppose a decision was made in a particular situation to choose task A from the set of tasks $\{A, B\}$, where both tasks achieve the goal G. When faced with the same (or a similar) choice in a similar future situation, a task analogous to A in that situation can be chosen, assuming that the choice of A was successful before.

When SEQER must decide which task in the Task Set to execute next, it searches for episodes in which similar decisions were made. If it finds such a DE, it can opt to choose the task in the current ("target") situation that is analogous to the task chosen in the previous ("source") episode. SEQER's Case Applier component retrieves DEs using the retrieval mechanisms of the CAPER.

Because this case memory is not preindexed, there is high flexibility as to which features of the current situation (i.e., Lab State, Goal Set, Hypothesis Set, current samples, etc.) can be used as part of the retrieval probe. For protein sequencing, SEQER currently uses a simple probe consisting of the type and object of each executable task in the Task Set, as shown in Figure 8.2. Other features of the sample, tasks, etc. could be added to the probe for finer discrimination.

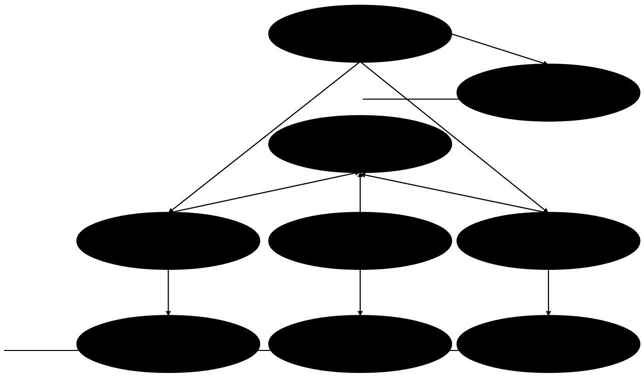

Figure 8.2: A sample SEQER retrieval probe graph.

In addition, SEQER can make use of CAPER's probe relaxation mechanism in the event that no cases match a retrieval probe (or some feature in the retrieval probe). Features in the retrieval probe can be relaxed using information from the concept is-a hierarchy. For example, SEQER may have to choose between the tasks Clostripain Cleavage and Acid Hydrolysis. It issues a probe containing these tasks and gets no matching cases. Using its taxonomic knowledge that Clostripain Cleavage is-a Enzymatic Cleavage Procedure and that Acid Hydrolysis is-a Hydrolysis Procedure, SEQER could issue a new, more general probe, replacing "Acid Hydrolysis" with "Hydrolysis" and/or replacing "Clostripain Cleavage" with "Enzymatic Cleavage".

Although protein sequencing is a fairly well-understood domain, it is a real-world domain of some complexity. The use of CAPER's mechanisms to represent and retrieve decision episodes facilitated the building of the SEQER prototype. The task of manual protein sequencing, however, may be more amenable to the use of schematic (abstract) knowledge for planning. The human experts tended to recall specific experiments only when the fairly routine procedure failed to provide guidance, e.g., when sequencing (parts of) complex proteins. The expert probably obtained this schematic knowledge of the high-level manual protein sequencing process through both instructional materials (the SequenceIt user manual, etc.) and induction over many similar experiments.

It would be interesting to explore the applicability of SEQER/CAPER in more complex, less well-understood scientific domains in which cases may play a greater role in planning experiments. In addition, more research needs to be done to determine and exploit the connection between CBP systems like CAPER that use some limited derivational information and systems like PRODIGY/ANALOGY that use full derivational information.

8.3 The Cognitive Science Connection

In addition to A.I. research in derivational analogy, ideas from the study of analogy and memory by cognitive scientists have inspired work in CBR/CBP in general and CAPER in particular. Cognitive science can provide both design goals and design ideas for application-oriented systems (i.e., systems that are not doing cognitive modeling but rather seek to solve real-world problems).[4] An example of a design goal inspired by cognitive science is a memory retrieval process in a CBR system that has the flexible, associative character of human analogue retrieval. Such a retrieval process should avoid some of the limitations of human analogue retrieval such as a heavy reliance on surface similarity criteria, etc. Examples of design ideas inspired by cognitive science include episodic memory representations (described in the next section) and the analogical reasoning process used by many computational systems

---

[4] The Josephsons would call these application-oriented systems an example of the "AI as engineering" approach, versus "AI as traditional science", in which hypotheses about human cognition are being tested (Josephson and Josephson, 1994). They also delineate another approach to AI which they call "AI as design science". In this approach, a system is studied as a "cognitive artifact" in its own right to discover principles that "hold for cognition in the abstract" (including, presumably, analogical reasoning) (ibid, 35).

(Section 8.3.2).

8.3.1 Episodic Knowledge in Human Memory

Human memory must support the operations of storage, retrieval, and adaptation. Cognitive scientists have studied human memory extensively (e.g., (Barsalou, 1992; Cohen, 1989; Cohen *et al.*, 1993)). This work has in turn inspired A.I. researchers to develop models of human memory in computers that can support various types of reasoning, including case-based reasoning (Kolodner, 1993, Chapter 4).

Human memory can hold a variety of kinds of knowledge including generalized concepts (e.g., chair, dog, eating, beauty) and specific instances of concepts (e.g, my desk chair, Snoopy). Knowledge about events is often called *episodic memory* (Tulving, 1983).[5] Episodic knowledge includes knowledge of specific events (e.g., my fourth birthday party) and generalized events (e.g, a child's birthday party). The latter kind of knowledge is induced from the former. Specific events may be reported or autobiographical and specific (e.g., seeing a particular play) or extended (e.g., going to college) (Barsalou, 1988). Autobiographical memories can include the participant(s), activity, location, date/time (period), causes, conversations, emotions, sensory impressions, etc.

Inspired by work on human episodic memory, Schank proposed several "data structures" to encode and organize event knowledge in memory for understanding and reasoning systems. A *script* encodes knowledge about stereotypical event sequences such as going to a restaurant (Schank and Abelson, 1977). Scripts have temporally-ordered scenes (e.g., ordering a meal, eating the meal, paying) and default roles and props. Several computer programs have used scripts in their reasoning (Schank and Riesbeck, 1981).

Seeking a data structure that could encode and organize more dynamic knowledge used by humans, including knowledge about particular events, Schank developed MOPs, based on frames. A *MOP* (memory organization packet) encodes knowledge about classes of events (e.g., taking a trip) and scenes common to several kinds of events (e.g., getting a ticket) (Schank, 1982). MOPs are linked to other MOPs using abstraction links, scene links, and index links which collectively form abstraction hierarchies, packaging hierarchies, and discrimination networks, respectively. Discrimination networks are used for indexing the set of MOPs into subsets for efficiency (see Section 2.3.2). MOPs are used in several case-based reasoning programs (Riesbeck and Schank, 1989) and in a computer implementation of autobiographical memory, CYRUS (Kolodner, 1984), which encodes events in Cyrus Vance's days as U.S. Secretary of State.

CAPER also uses structured representation for episodic (event) knowledge (Section 3.4). While the representations of episodic knowledge were engineered specifically for CAPER, they turn out to resemble those used to represent events in a particular theory of autobiographical memory motivated by cognitive psychology experiments (Barsalou, 1988). CAPER's episodes, however, are not stored chronologically as are events in Barsalou's model.

---

[5] There is some debate about whether episodic and semantic memory are actually distinct in humans (Barsalou, 1988).

Retrieval Descriptions in Human Memory

As described in Chapter 4, CAPER uses a type of data structure called a "retrieval description". This data structure specifies the kind of information (case) sought from memory. The idea of using a retrieval description is partially based on Norman and Bobrow's work in the psychology of memory (Norman and Bobrow, 1979). Their theory of human memory retrieval includes the notion of a "description" that guides the memory search process. A description can contain one or more "perspectives", views of the target item with respect to a particular purpose or goal (similar to the the "perspective" construct in frame-based, knowledge representation language KRL (Bobrow and Norman, 1977)).

A description is formed then matched to memory. The items retrieved are evaluated for selection. The description formation and evaluation process can also require retrieval of conceptual knowledge. The effectiveness of a description is a function of its discriminability (the ability to pick out relevant information in memory from irrelevant information) and constructibility (the likelihood that an appropriate description will be constructed at retrieval time).

The proposed "planning to retrieve" process in CAPER (Section 9.1.3) would extend the capabilities of CAPER for forming retrieval descriptions. This process was also inspired by work in psychology on human memory retrieval.

8.3.2 Analogical Reasoning

Components for retrieval and adaptation need to be combined with a control component that directs the CBP process. The CBP/CBR process can be conceived of as a specialization of the general analogical reasoning process. The conception of the analogical reasoning process presented in this section was used in the design of CAPER.

The needs and abilities of people to perform various reasoning tasks are important to consider when building A.I. applications. Cognitive scientists have long been studying how people use analogy (e.g., (Vosniadou and Ortony, 1989; Hesse, 1966)). Their work has informed that of A.I. researchers building CBR systems and vice versa. In addition to psychological experiments, several computer models have been developed to test the theories of human analogical reasoning (e.g.,(Falkenhainer *et al.*, 1990; Holyoak and Thagard, 1995)). The performance of these computer models is then compared with that of people. If the performance is similar, the theory embodied in the computer model can be judged a sufficient (though not a necessary) explanation for the particular human reasoning behavior.

The analogical reasoning task can be characterized for both people and computers as consisting of the following phases which occur when the reasoner is given a target analogue — a new problem to solve or situation to understand (Hall, 1989):

1. **Retrieve** (recall, recognize) a source analogue
2. **Elaborate** an analogical mapping between the source analogue and target analogue and transfer knowledge between the analogues.

3. **Evaluate** the elaborated analogy
4. **Consolidate** the information generated using the analogy

In human analogical reasoning, these phases are not so much sequential steps as (possibly recurring) activities typical of an episode of analogical reasoning.

In the *retrieval phase*, a source analogue that is similar to the target analogue must be retrieved from a set of potential source analogues.

In the *elaboration phase*, a mapping is created and knowledge is transferred. A *mapping* is a set of correspondences between elements of the source analogue and elements of the target analogue. A *local mapping* is a mapping between elements of two analogues. A *global mapping* is a mapping between two analogues. In the "atom is like a solar system" analogy, there is a global mapping between atom and solar system. A global mapping is a set of local mappings. For example, in the atom–solar system analogy, there are several local mappings: nucleus maps to sun, electron maps to planet, etc. Some elements of the source or target analogue may be unmapped: e.g., comet, quark, etc. The atom–solar system analogy is an example of a *cross-domain analogy* in which the target analogue belongs to one *domain* (e.g., pre-quantum physics) and the source belongs to another (e.g., astronomy).

In addition to objects, relations – consisting of a predicate and arguments – can also be mapped, e.g., *revolves-around(Electron,Nucleus)* can be mapped to *revolves-around(Sun,Planet)*. Higher-order relations – relations with at least one argument that is itself a relation – can also be mapped: e.g.,
cause(attracts(Nucleus,Electron)), could be mapped to
cause(attracts(Sun,Planets)). This ability to map higher-order relations appears to be central to human analogical reasoning competence (e.g., (Gentner, 1989; Holyoak and Thagard, 1995)).

Mappings suggest *analogical inferences*. For example, the atom–solar system analogy can be useful to a student trying to learn about atoms using what she already knows about solar systems. It could also be useful to a scientist trying to predict new phenomena about atoms given what is known about solar systems. In both cases, knowledge *transfer* can occur from the source (solar system) to the target (atom) (and perhaps vice versa). This transfer process results in analogical inferences being made about the target domain. For example, given a mapping from planet to electrons, we might infer (incorrectly) that electrons, like planets, are rocky.

In the *evaluation phase*, analogical inferences are verified. Analogical inferences may be incorrect, as the one above is, because they depend on the degree of similarity between the source and target domains. Like inductive inferences, they must be verified by some other means (e.g., proven deductively, verified empirically, etc.). If too many spurious analogical inferences result from a particular analogy, the analogy itself may be judged poor.

In the *consolidation phase*, the "products" of the analogy are stored in the reasoner's memory. These products include the the analogy itself (target and source analogues and the mapping between them), verified analogical inferences, and any other knowledge that has been learned using the analogy (e.g., schemata/abstractions induced from both analogues, etc.).

8.4 Chapter Summary

This chapter discussed some of the methodology used in the design and development of CAPER's mechanisms and the extension of these mechanisms to support derivational analogy in SEQER. The design methodology included taking inspiration from psychological theories of human memory retrieval and analogical reasoning.

Chapter 9

Conclusion

This chapter describes some future directions for CAPER and summarizes its contributions.

9.1 Future Directions for CAPER

Future work for CAPER includes continuing the ongoing evaluation of the prototype in the UM TRANSLOG domain to obtain additional quantitative results about CAPER's current mechanisms and extending these mechanisms to provide new capabilities. Some ideas about evaluation and extension are described briefly in the following sections.

9.1.1 Additional Evaluation

Techniques for empirically evaluating CBR systems are a relatively new subject of study in the research community. Only recently was the first workshop held on evaluating CBR systems.[1]

Cohen distinguishes several kinds of evaluation for AI (planning) systems (Cohen *et al.*, 1994):

- "Paradigmatic comparison" is the evaluation in isolation of a planning system embodying a new approach. The goal is to assess the benefits of the new approach.
- "Alchemical comparison" is the comparison of a planning system to its previous versions. The goal is to assess the benefits of new functionality added to the system.
- "Scientific comparison" is the comparison of a planning system to others in the field. The goal is to assess the benefits of a system over its "rivals".

Some ideas about how Cohen's framework could be applied to CBP systems are given in (Kettler and Hendler, 1994). Here a few of these ideas will be discussed briefly. Further empirical evaluation of CAPER will be useful to confirm the predictions of its performance

---

[1] The Case-based Reasoning Workshop held at AAAI-94 in Seattle.

based on a (qualitative) analysis of its design and the qualitative observations made so far of it in action.

CAPER's evaluation to date has been primarily of the "paradigmatic" sort to determine how well its methods work. This includes the experiments done to assess the scalability of query times for typical queries (Section 4.7) and those done to determine how much CAPER uses memory (Chapter 7). These experiments assess CAPER's ability to overcome the "serial retrieval bottleneck" found in many CBR systems. Further experiments are needed to determine adaptation efficacy and efficiency. The design of such experiments is complicated by a CBR system's performance being dependent on the contents of its casebase.

"Alchemical" comparison could eventually be done to determine how future extensions to CAPER, such as the use of more knowledge-intensive search planning techniques described below, improve its planning performance and at what cost. "Scientific" comparison would also be useful. Here CAPER would be compared to other reuse-only CBP systems. This would involve finding a common platform and planning domain (casebase) for CAPER and the other CBP systems. The UM TRANSLOG domain (Section A.1) may be one such domain. Comparison across several planning domains which have varying levels of domain theories, etc. is preferable. Metrics could include plan length, casebase access cost, knowledge-engineering effort for the casebase, etc. Based on its design and observations of its performance to date, CAPER should do well against other reuse-only CBP systems. Some of the techniques for evaluating exemplar-based machine learning systems (e.g., (Shavlik and Dietterich, 1990; Aha *et al.*, 1991)) may aid in scientific comparison between CBR/CBP systems, provided some benchmark test domains could be created.

9.1.2 Retrieval

CAPER currently uses a flexible search planning mechanism to form and modify retrieval probes. This could be extended, as described in the next section. CAPER's retrieval mechanism is also being investigated for use in a knowledge discovery system being built for a medical KB to be used by medical doctors.[2] This KB was created in PARKA from 20,000 database records. A domain concept ontology was added to support queries of varying specificity by a doctor browsing the KB to discover new and interesting relationships among patient cases (e.g., recurring structures, etc. not easily observable using only the original database).

In addition to changes to CAPER, some enhancements to the PARKA Structure Matcher may be useful. For example, CAPER could pass an effort upper bound along with a retrieval probe to the PARKA Structure Matcher. When the query processing effort exceeds this bound, the control would be returned to CAPER along with the partial result, etc. This would tell CAPER that it has formulated a potentially "bad" query (such as one that will return too many plans, each of which must then be checked for similarity, etc.). CAPER could then reformulate the query.

---

[2] This KB is being constructed using PARKA by Merwyn Taylor, under the direction of Prof. James Hendler.

9.1.3 Search Planning

Another means of exploiting the flexibility of CAPER's retrieval methods is to treat the case/plan retrieval task (described in section Chapter 4), specifically the formation of retrieval descriptions, as a full-blown planning task. This would involve augmenting the search planning (automated formation of retrieval descriptions and retrieval plans) that CAPER currently does (Section 4.6) with additional planning techniques and knowledge.[3] This proposed "planning to retrieve" approach is motivated by (1) the difficulty in determining a priori what features should be included in the initial and subsequent retrieval descriptions and (2) results from psychology, discussed below, showing the benefits for recall when explicit retrieval plans are used.

A CAPER retrieval description initially contains all the features in the target problem description. If this retrieval description is unsuccessful, it can be modified in two ways. One way involves the dynamic application of retrieval description operators to search a space of retrieval descriptions to construct a retrieval plan from scratch, as described in Section 4.6. This provides more flexible plan retrieval that is useful domains such as UM TRANSLOG in which the target problems may vary widely.

The other way for forming retrieval descriptions involves following a fully-specified retrieval plan, a sequence of retrieval descriptions. A fully-specified retrieval plan is currently supplied by the domain engineer. This is done for plan retrieval in the Car Construction domain. This approach is less flexible but requires no search of the retrieval description space and is viable in certain kinds of domains. These include those in which the initial situations do not vary much. In the Car Construction domain, for example, the initial situations are very similar to each other (same set of parts, etc.).

Treating retrieval plan formation as a "full-blown" planning task would allow both generative and case-based planning methods to be employed. other planning techniques. The proposed core "planning to retrieve" process is shown in Figure 9.1. Previous plans used to retrieve cases in the same (and similar) domains could be used for case-based planning. Domain-specific knowledge could be used to guide generative planning. This includes the use of retrieval strategies, heuristics for forming and modifying retrieval descriptions. Such strategies are similar to automatic query reformation in database systems and information seeking strategies in text-based information retrieval systems (e.g., (Belkin and Cool, 1993)).

There is also evidence from the naturalistic studies of long-term memory, particularly autobiographical event memory, that humans can successfully employ retrieval strategies and retrieval plans in a range of retrieval tasks. A compelling experiment by Williams and Hollan showed subjects' extensive use of retrieval strategies to modify retrieval descriptions for the task of recalling their high school classmates' names (after 4-19 years) (Williams and Hollan, 1981) . The strategies included:

- extended retrieval in which additional cues about the target item are (recursively)

---

[3] Recall that the search planning task is is distinct from the overall planning task (i.e., solving the target planning problem) in which it is embedded.

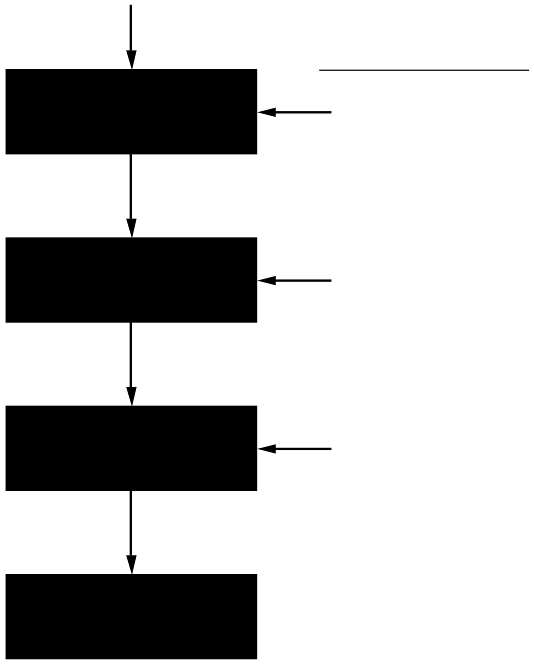

Figure 9.1: Proposed retrieval description formation process using retrieval strategies and other knowledge and planning techniques

retrieved: e.g., the target person's first name was Art, he was in subject's 10th grade art class, he played bass guitar, etc.

- systematic hypothesizing of cues, e.g., "his first name began with a 'D'...Don, Dave, Douglas..."
- inferring cues
- choosing then searching a context, e.g., "Who was in my P.E. class?", "Who was on our football team?" "Who do I remember from the cafeteria?", "Who was a friend of Bill's?"

As shown in Figure 9.2, the generic (domain-independent) retrieval strategy "Find a context" could be instantiated for the high school classmates task as "Find names of classmates associated with locations in a/your high school", which, using additional conceptual or episodic knowledge, decomposes into the subtasks "Find names of classmates associated with the library", "Find names of classmates associated with the cafeteria", etc. (each of these being a new retrieval description).

Anderson and Pichert found that people used schema-like knowledge to recall additional information from a story they had read (and encoded) from another perspective (Anderson

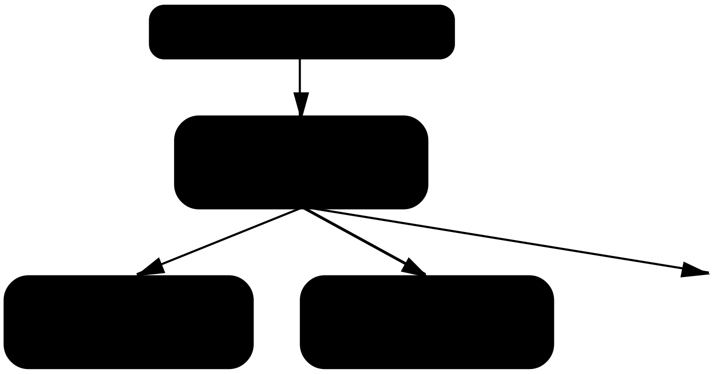

Figure 9.2: Application of the "find a context" retrieval strategy to generate retrieval descriptions

and Pichert, 1978). Thus they argue that the ways in which information can be retrieved is not determined solely by its encoding context and can involve the use of conceptual knowledge to establish and search a context (using descriptions). Norman and Bobrow argue for the use of descriptions and contexts in retrieval (Norman and Bobrow, 1979). Reiser (Reiser *et al.*, 1986) and Linton (Linton, 1986) also present evidence for recall plans and strategies based on protocol analysis.

CAPER could make use of similar knowledge-intensive retrieval strategies. This is another example of the possible synergy between cognitive science and CBR/CBP discussed in the Section 8.3. In CAPER, the application of retrieval strategies would entail the (recursive) retrieval of conceptual and/or episodic information from memory. Multiple retrievals, however, are cheap in PARKA. Retrieval strategies could be selected using the following information:

- the problem description (initial situation, goals, etc.)
- information from the target plan (validations, etc.)
- previous retrieval descriptions and their results from the PARKA Structure Matcher
- conceptual/episodic knowledge
- metamemory information
- previous retrieval plans/episodes (for case-based strategy selection)

9.2 Adaptation

Several avenues of exploration derive from CAPER's approach to plan adaptation. One enhancement would be the inclusion of plan validation structure as part of the retrieval probe to retrieve a patch/substitute plan. Currently only the target subgoals are included in the probe. The validation structure of the retrieved and chosen source plan is checked subsequently. Validation structure information could be included in the probe to match source plans that both achieve the target goals and minimize disturbances to the overall target plan (and its validation structure). This is currently not done because the PARKA Structure Matcher's requires an exact match to the retrieval probe. In addition, it is difficult to express "does not disturb validations x, y, and z" in the query language (e.g., several action effects might clobber a given validation, etc.).

Another enhancement to CAPER would be to fix a plan fault by using an alternate source plan, previously retrieved, or using an alternate mapping for a particular source plan. It might be better to use an alternate mapping, or even another source plan, if a particular mapping/plan is causing too many applicability failures. Because the FCS search tree and open list are preserved, the Plan Mapper can be reinvoked to return another mapping for a plan *previously* mapped. This might occur when a newly instantiated target plan has a predicted or actual failure due to an improperly filled role. An improperly filled role might be the result of finding no value for some plan variable that satisfies *all* of the plan variable constraints. An extension to CAPER would be to add the method "find an alternate reinstantiation for plan variable v filling role r" to its store of plan modification methods. A mechanism would need to be added to the Plan Modifier to determine when it was appropriate to use an alternate source plan or mapping, versus some other kind of refit.

A generative planner such as UMCP (Erol, 1995) could be integrated with CAPER, much as NONLIN was integrated with PRIAR. A generative planner could be used for refit tasks, either in addition to CAPER's current process of going back to memory or instead of it. A generative planner uses abstract knowledge (e.g., an operator schema) of how to reduce a goal. CAPER uses episodic knowledge when reducing a goal. In the UM TRANSLOG domain, this episodic knowledge was originally derived from cases seeded by UM NONLIN, which uses operator schemas. Hence the distinction between generative refit and reretrieval from the casebase blurs somewhat. Ideally operator schemas would be induced from episodes.

Given the availability of both episodic knowledge and abstract knowledge, the planner would have to decide whether to use one kind or both. Episodic knowledge can be highly detailed (which is desirable, see (McCartney and Sanders, 1990)) but can also be idiosyncratic and hence potentially misleading when reused. Additional research is required to determine the benefits of each type of knowledge with respect to the planning mechanisms available, the kind of domain, etc.

CAPER's mechanisms could be used to repair failures found during plan execution. This would require a way of getting feedback into CAPER from (simulated) execution. This feedback in turn would be used to find failures not previously detected during annotation verification. This would be particularly useful in the event that the source plan's validation structure was incomplete (perhaps due to hand-encoding), and thus an applicability failure would not be detected prior to execution. Feedback from the real world is also useful to detect an inconsistent or incorrectly encoded domain model. Because CAPER can represent the structural relationships and dependencies among plan components, it is possible that a case-based approach could be used for failure diagnosis (such as SWALE (Schank, 1986) does, for example).

9.3 Contributions

This dissertation has presented a description of CAPER, a domain-independent CBP system. CAPER's memory and components for plan retrieval, plan elaboration, and plan adaptation were described, along with examples from two of CAPER's test domains, Car Construction and UM TRANSLOG. Some of the results from cognitive science research on human analogical reasoning and memory that have inspired the CBP/CBR approach in general, and CAPER's in particular, were presented.

CAPER's approach was motivated by surveying some CBP systems. These systems were classified on a spectrum of plan reuse. Reuse-optional CBP systems, like generative planning systems, can produce a plan from scratch. Starting from the null plan, they can find a solution plan for the target problem, if their mechanisms are sound and complete. They can also reuse a previous plan when solving a new problem, thus reducing search and achieving a speedup benefit over planning from scratch. Reuse-optional systems include PRIAR (Kambhampati and Hendler, 1992), SPA (Hanks and Weld, 1995), and PRODIGY/ANALOGY (Veloso, 1992).

Reuse-only CBP systems cannot produce plans from scratch. They rely on reusing previous plans and on more limited adaptation mechanisms, which are often ad hoc or heuristic. These more limited adaptation mechanisms have the benefit of requiring less strong domain causal knowledge than those of generative-only and reuse-optional planning systems. This allows a system to function in less well-understood domains where a strong domain theory is a lacking. The limited adaptation mechanisms are adequate if the source plan being adapted is close to the solution plan. Reuse-only CBP systems include more "traditional" CBP systems such as CHEF (Hammond, 1986).

Traditional, reuse-only CBP systems have at least four major requirements to produce good plans. Unfortunately, many of these systems have fallen short of meeting these requirements. These shortcomings and the availability of a fast, scalable, and parallel memory via PARKA (Evett *et al.*, 1994; Kettler *et al.*, 1995) have motivated the approach taken in CAPER. The major requirements for CBP systems in general, and reuse-only CBP systems in particular, are:

- **support for large casebases**

  A large casebase allows better coverage of the problem domain. Better coverage increases the likelihood of finding a plan in the vicinity of the target plan. (i.e., the plan space is less sparse). Large casebases require scalable retrieval methods and automated case encoding methods.

Many CBP systems use very small casebases, often hand-encoded or hand-indexed. CAPER's Casebase Seeder can automatically seed an arbitrarily large casebase. The parallel retrieval methods of PARKA scale well to large casebases containing hundreds of cases (hundreds of thousands of frames).

- **an expressive case representation**

  CBP systems benefit when starting from a source plan that is close to the target plan. Reuse-only systems require this, and reuse-optional systems derive a speedup from it. Finding a source plan close to the target requires matching and encoding cases at a fine level of detail and the ability to capture structural relations in an expressive case representation scheme.

  Many CBP systems have used less expressive, feature-based representations. CAPER uses an expressive, graph-structured case representation to capture details and structural relations among objects in planning problem and dependencies in plans.

- **effective and efficient retrieval mechanisms**

  Flexible retrieval mechanisms are needed to match cases at a fine level of detail to find a source plan that is close to the target plan. Retrieval must also be efficient and scalable to support large (growing) casebases, especially if a speedup benefit from CBP is a goal.

  Most traditional reuse-only systems employ the technique of pre-indexing to improve retrieval efficiency at the expense of retrieval-time flexibility and the general-purpose utility of the casebase. In CAPER flexible queries can be made of memory that contain any features of the target problem. As memory is not preindexed, a source plan (or part of a plan) can be retrieved through any of its constituent features. Queries can be formulated and modified automatically.

- **effective and efficient adaptation mechanisms**

  Because a casebase tends to be sparse relative to the entire problem domain, some adaptation mechanisms are needed to transform an old plan into a solution for the target problem.

  The adaptation methods used by traditional reuse-only CBP systems have often been more ad-hoc and less powerful. The retrieval and mapping mechanisms in CAPER are more sophisticated than those of many reuse-optional systems. CAPER incorporates adaptation methods based on those used in generative planning systems into a reuse-only framework. These methods include the use of plan validation structures (based on those of PRIAR) and the composition of subplans. With its flexible retrieval mechanisms and case representation scheme, CAPER can go to memory often to get patch/replacement (sub)plans.

Though CAPER's techniques have the greatest potential for rich, real-world domains that require expressive case representations and flexible retrieval, the benefits of CAPER's approach have been illustrated in a benchmark domain, UM TRANSLOG. Components of CAPER have also been used for derivational analogy in a protein sequencing experiment planning domain. Further analysis and empirical of CAPER's techniques is needed to assess their quantitative benefits in terms of planning time, solution quality, etc.

The contributions of CAPER are improvements over the traditional reuse-only CBP systems by addressing the limitations of such systems listed above.

- CAPER **takes a more memory-intensive approach to CBP.**

  The speed of the underlying parallel retrieval methods of PARKA is exploited by going to memory often during all phases of planning: retrieval, elaboration, and adaptation. During retrieval, memory is accessed repeatedly with automatically modified retrieval descriptions until a probe succeeds. During elaboration, memory is accessed to check plan variable constraints. During adaptation, memory is accessed to check plan validations and to find substitute or patch plans.

  CAPER's approach is also memory-intensive in the sense of "much memory", i.e., the use of large casebases. These casebases can be automatically seeded via CAPER's Casebase Seeder component.

- CAPER **has more flexible retrieval.**

  To find a plan from a case that better matches the target case, CAPER allows matching at a fine level of detail. Details of a case, including structural relations of physical objects and dependency relations (plan validations) of plan actions, can be stored in CAPER's expressive, graph-structured case representation, a semantic network. A retrieval description (retrieval probe) can include any features of the target problem for determining surface, structural, or pragmatic similarity. As memory is not pre-indexed, a source plan (or part of a plan) can be retrieved through any of its constituent nodes.

- CAPER **has more principled plan adaptation methods.**

  To adapt a source plan to the target problem, CAPER first reinstantiates the source plan by mappings its plan variable bindings to the target case. Plan validations are used as constraints on these bindings. Differences between the old and new cases that cause validation structure applicability failures are detected by an annotation verification process based on that of PRIAR (Kambhampati, 1989). Lacking a generative planner, CAPER goes back to memory to retrieve a patch or substitute plan. This approach integrates plan adaptation techniques from generative planning into a reuse-only CBP framework.

CAPER's techniques for graph-structured case representation, automatic casebase seeding, search planning, and mapping and adaptation using plan validations benefit from the underlying high-performance retrieval mechanisms of PARKA which provides speed and scalability. These techniques could also be used to improve planning when only serial retrieval mechanisms are available. Conversely, providing a traditional reuse-only CBP system with a fast memory would not automatically improve retrieval flexibility, provide a more principled approach to plan adaptation, or reduce domain engineering effort.

In summary, CAPER contributes a novel approach to reuse-only case-based planning. It is hoped that these techniques will begin to bridge the gap between research CBP systems and those that solve problems in the real world.

Appendix A

CAPER's Test Domains

This section describes the domains being used to develop and test the CAPER prototype. The primary domain being used is a transport logistics domain, UM TRANSLOG. The Car Construction domain used to test earlier versions of CAPER is described briefly in Section A.2. As was discussed in Section 8.2, mechanisms from CAPER have also been used for planning scientific experiments for protein primary structure determination by derivational analogy (Kettler and Darden, 1993).

A.1 Transport Logistics Domain

The primary domain developed to test CAPER is a transport logistics planning domain, UM TRANSLOG. An overview of the domain is presented here. The following description is largely taken from (Andrews *et al.*, 1995), which has additional details.

A.1.1 Motivations

Initially, a number of simple toy domains have been devised to assist in the analysis and evaluation of planning systems and techniques. The most 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 and has been made available to the A.I. planning community. In addition to providing a domain useful for benchmarking, an additional design goal was that the domain be encodable for both UM NONLIN (Ghosh *et al.*, 1992) and CAPER. A UM NONLIN encoding of this domain is used when automatically seeding a CAPER casebase (Section 3.5).

UM TRANSLOG was inspired by a transport logistics domain developed at CMU (Veloso, 1992) but is an order of magnitude larger in size (41 actions versus 6), number of features, and types of interactions than the CMU domain. It provides a rich set of entities, attributes, actions and conditions, which can be used to specify rather complex planning problems with a variety of plan interactions. The detailed set of operators provides long plans with many possible solutions to the same problem, and thus this domain can also be used to evaluate the solution quality of planning systems.

In the UM TRANSLOG domain, the planner is given one or more goals, where a goal is typically the delivery of a particular package from an origin to a destination. Given our design goal of creating a domain more complex than toy domains such as blocks world, we modelled additional aspects of transport logistics not in the CMU Transport Logistics domain. 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

Our goal, however, was *not* to model "real" transport planning as described by a domain expert, for example. Obviously the kind of transport logistic planning done in the "real world" by UPS, the military, etc. is considerably more complicated, particularly as the world state is highly dynamic and often not fully known by the planner. Some major aspects of transport logistics planning we are *not* modelling in order to keep the domain "manageable" and to stay with in the capabilities of the planning systems we are testing include:

- package sizes and vehicle capacities (We assume unlimited vehicle capacity.)
- numeric distances between locations[1]
- temporal aspects including delivery deadlines, action durations (travel time, etc.), action concurrency, etc.

---

[1] Distances are "simulated" to some extent by the grouping of locations into cities and cities into regions and the explicit specification of intercity routes.

A.1.2 Entities

UM Translog entities include individual locations (cities, etc.) and individual objects (routes, vehicles, equipment, and packages). Each entity is described by a constant symbol (e.g., "Truck-1", "Package-2") and one or more predicates that are asserted by a user (in the initial state given to the planner) or by the effects of instantiated plan actions. Each entity has a primary type. Primary entity types include location types, route types, vehicle types, equipment types, and package types, described in the following sections.

Location Types

As shown in Figure A.1, 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*.

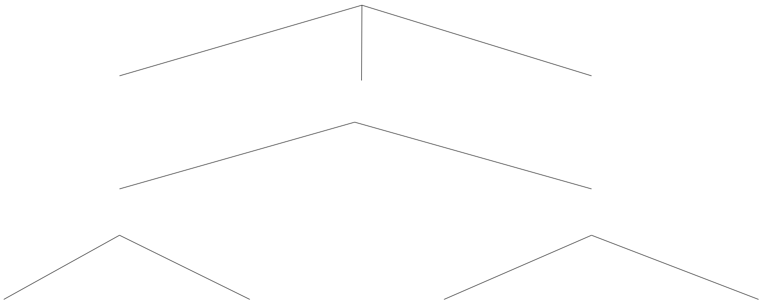

Figure A.1: Location type hierarchy

Regions contain one or more cities.

Cities contain one or more city locations (specified via the predicate *in-city*). Some cities are compatible with hazardous packages.

A city location is located in a specific city.

A transport center can be used for air/rail direct and indirect transport (see Section Section A.1.4). Transport centers can optionally be specified as transport hubs. Hub transport centers can be used for indirect transport. Transport centers serve specific cities. 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. A customer location is located in a specific city.

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

Routes Types

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 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. Note that routes are directional: traffic flows from the origin to the destination. Routes have an availability status. For example, an particular road route may be temporarily unavailable due to construction.

Vehicle Types

Primary vehicle types are *truck*, *airplane*, *train* (train engine), and *traincar*. Trucks and traincars have subtypes: a single physical subtype and an optional specialty subtype. Physical truck/traincar subtypes are:

Physical Subtype	**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:

Specialty Subtype	**Examples**
refrigerated	refrigerated truck/traincar, etc.
armored	armored truck/traincar, etc.

Specialty subtypes cannot be specified if the truck or traincar has a physical type of *mail*, *livestock* or *auto*. The physical subtype of a truck, traincar, or airplane must be compatible with the physical subtype of a package. When specified, a package specialty subtype and truck/traincar specialty subtype, must also be compatible.

Vehicles of type *train* — 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.

Equipment Types

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 and status.

Package Types

Packages have type *Package*. Packages have subtypes: a single physical subtype and, optionally, one or more specialty subtypes. Physical package subtypes are:

Physical Subtype	**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:

Specialty Subtype	**Examples**
perishable	produce, frozen food, etc.
hazardous	petroleum, nuclear waste, etc.
valuable	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. The following table lists compatible package and vehicle types:

Package Physical Subtype	**Vehicle (Sub)type**
regular	regular
bulky	flatbed
liquid	tanker
granular	hopper
perishable	refrigerated
valuable	armored
mail	mail
livestock	livestock
auto	auto
regular	airplane
mail	airplane

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.

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

A.1.3 Actions

Actions for Loading/Unloading

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 *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 the 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* involves the 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)*.

Transit Actions

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.

A.1.4 Tasks

Tasks are the jobs to be planned for. Each task is either posed by the user to the planner as part of a planning problem, or it is introduced by a planner as subtasks if it serves to accomplish a task posed by the user. In this aspect tasks are similar to goals in STRIPS-style planning, only with richer structure. Figure A.2 presents the organization of tasks and their subtasks, which are discussed below. For details, see the task/method definitions (available from the WWW site or via FTP) for the particular planner to be used.

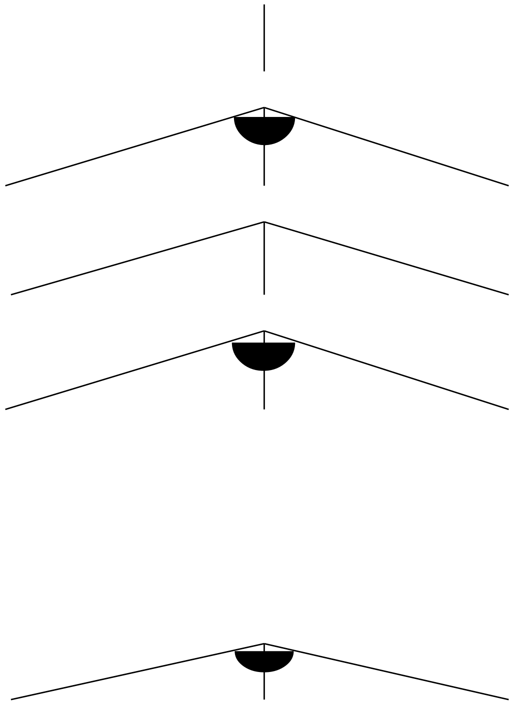

Figure A.2: Top-level task hierarchy

Task *AT-PACKAGE(package destination)* requires transporting the package to its destination.

Task *TRANSPORT(package origin destination)* involves picking up the package, carrying it to its destination, and delivering it, provided that the package is currently at *origin*.

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

Task *CARRY(package origin destination)* moves a 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 A.3 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.

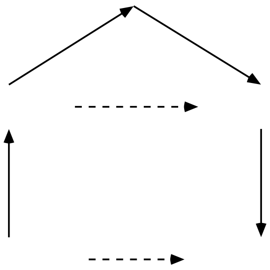

Figure A.3: Transport paths

Task *CARRY-DIRECT(package origin destination)* 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.

Task *AT-VEHICLE(vehicle destination)* moves a vehicle of type truck, plane, or train, 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. When a vehicle moves, so do any packages it contains.

Task *LOAD/UNLOAD(package vehicle location)* involves 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.

A.2 Car Construction Domain

The first domain developed to test CAPER was a simplified car construction (assembly) domain. This is a "toy" domain constructed initially to develop an understanding of the problem. The simplifications made also allow the automated generation of seed cases using the UM NONLIN planner (Section 3.5). The goal in this domain is to generate a plan for assembling a particular kind of car: e.g., a hard-top sedan with a 4-cylinder engine, sunroof, and radio with cassette unit. The initial situation is a set of parts. The car is to be built in a particular factory by a robot who can assemble the pieces in varying order.

This domain was abandoned after early development of CAPER for several reasons. This domain had plans that were highly modular and thus subplans from different original plans could be merged with little effort. This unfortunately is not a characteristic of many more realistic domains, where interactions among components frequently arise and must be handled. The problem space was fairly small and the domain itself not very "rich" — i.e., a small number of actions, relatively short plans, few constraints on planning, identical initial situations for planning problems, etc. To provide a richer testbed for CAPER, the UM TRANSLOG domain described in the next section was developed. Examples of CAPER using the Car Construction domain can be found in (Kettler *et al.*, 1994).

Appendix B

Details of Data Structures

B.1 Retrieval

B.1.1 Case Features

A case feature f includes the following information:

- **id**: a unique identifier for f
- **class**: the feature class that f is an instance of
- **origin**: the feature that f was derived from and the method of derivation
- **facets**: a set of feature facets
- **query forms**: a probe graph template or macro pattern for PARKA Structure Matcher queries
- **score functions**: (optional) functions for evaluating the goodness (badness) of a match (or lack of a match) to f
- **generalizations**: (optional) superclasses of f that can be used by an retrieval description operator for generalizing (or specializing) f

Features have predefined, domain-specific feature classes. A feature class provides default values to its instances, which are individual features. A feature class also has associated methods for modification of its instances via retrieval description operators.

B.1.2 Case Feature Facets

A feature facet describes an individual object or relationship that is involved in the particular case facts represented by a case feature. A feature facet f has the following information:

- **id**: a unique identifier for f

- **name**: the name of f
- **class**: the feature facet class that f is an instance of
- **facet type**: "variable" or "constant". A constant facet has a fixed value.
- **feature**: the case feature that f is a part of
- **origin**: the facet that f was derived from the and the method of derivation
- **value**: the (current) value of f
- **variable**: the query variable for f (variable facets only)
- **variable types**: a type constraint on the value of f (variable facets only). A variable type is $\langle p_i, c_i \rangle$, where p_i is a predicate, c_i is a category (frame). Variable types correspond to unary constraints on the facet value v: $\forall i, p_i(v, c_i)$ must hold. An ordered list of generalizations g_i can be specified for c_i.

Feature facets have predefined, domain-specific feature classes. A feature facet class provides default values to its instances, which are individual feature facets. A feature facet class can also have associated methods for modifying a facet via an retrieval description operator (variabilizing or generalizing the facet, etc.).

B.1.3 Problem Description

A problem description d for problem p contains the following information:

- **id**: a unique identifier for d
- **subproblems**: a set of problem descriptions for p's subproblems (subgoals)
- **superproblems**: a set of problem descriptions for p's superproblems (parent goals)
- **features**: a set of case features
- **feature mappings**: a set of $\langle$target feature,source feature$\rangle$
- **initial situation**: the frame representing the p's initial situation
- **target plan**: the target plan descriptor for the current target (sub)plan to solve p
- **target plans**: a set of target plan descriptors of all target (sub)plans to solve p (and its subproblems)
- **source plans**: a set of source plan descriptors for chosen source (sub)plans
- **retrieval plans**: an ordered list of retrieval plans

- **HTN**: the hierarchical task network (Section 3.4.3) for the overall target plan (a partially-ordered set of target node descriptors)

For example, problem descriptor *PDESC-3* describes the top-level target problem of *Case-Make-Deliveries-3* after plan retrieval, mapping, and instantiation and prior to plan checking:

```
Target Problem PDESC-3
   Case: Case-Make-Deliveries-3
    Rtvl. Plans: (RPLAN-1)
   Source Plans: (SPLAN-1)
   Target Plans: (TPLAN-1)
     Root Plan: (TPLAN-1)
    Subproblems: NIL
   Features: (FEATURE-3.1)
   Mappings: ((FEATURE-3.1 . FEATURE-3.1S))
        HTN: (HTN-1)
```

B.1.4 Retrieval Descriptions

A retrieval description r contains the following information:

- **id**: a unique identifier
- **features**: a set of case features
- **retrieval plan**: the retrieval plan that r is an "action" of
- **origin**: the retrieval description that r was derived from and the retrieval description operator (or methods) used to derive it
- **query forms**: the PARKA Structure Matcher query forms
- **query variables**: a set of $\langle$ facet variable, query variable $\rangle$
- **query status**: the status of r: *unexecuted*, *success*, *no-match*, etc.
- **query bindings**: the sets of (query) variable bindings returned by a successful PARKA Structure Matcher query. A set of variable bindings is a set of $\langle x, v \rangle$, where x is a query variable and v is its value.

```
Rtvl. Description RDESC-1
   Derived-from: NIL
    Derived-via: :AUTO
     Rtvl. Plan: RPLAN-1
    Query Forms:
```

```
 ((instanceOf ?PKG-PVAR Plan-Variable)
  (var-Value= ?PKG-PVAR ?PKG) (thing= ?GOAL ?PKG-PVAR)
  (instanceOf ?DEST-PVAR Plan-Variable)
  (var-Value= ?DEST-PVAR City1-Cl2) (location= ?GOAL ?DEST-PVAR)
  etc.)
Query Vars.: ((ISIT . ?ISIT) (DEST . City1-Cl2)
     (ORGN . City1-Cl1) (PKG . ?PKG) (GOAL . ?GOAL) (PLAN . ?PLAN))
Query Status: :NO-MATCH
Query Bindings: NIL
Features: (FEATURE-3.1)
```

The query forms used to create the PARKA Structure Matcher query are shown. The query was unsuccessful and no bindings were returned.

B.1.5 Retrieval Description Operators

An example of the application of a retrieval description operator is given in Section 4.2. A retrieval description operator *o* has the following information:

- **id**: a unique identifier
- **class**: the predefined class *c* that *o* is an instance of. Class *c* is a subclass of class RDESC-RDOP, FEATURE-RDOP, or FACET-RDOP.

A retrieval description operator of class RDESC-RDOP modifies a retrieval description by adding or deleting features from it. It has the following information in addition to that above:

- **retrieval description object**: the retrieval description to be modified
- **old case feature**: the case feature to be deleted from retrieval description
- **new case features**: the case features to be added to the new retrieval description

A retrieval description operator of class FEATURE-RDOP, a subclass of RDESC-RDOP, modifies an retrieval description by modifying one of its features. It has the following information in addition to that of class RDESC-RDOP above:

- **case feature object**: the case feature *f* to be modified
- **old feature facet**: the old feature facet to be deleted from *f*
- **new feature facet**: the new feature facets to be added to *f*

A retrieval description operator of class FACET-RDOP, a subclass of FEATURE-RDOP, modifies an retrieval description by modifying one of its feature's facets. It has the following information in addition to that of class FEATURE-RDOP above:

- **feature facet object**: the feature facet f to be modified
- **old facet variable type**: the old facet variable type to be deleted from f
- **new facet variable type**: the new facet variable types to be added to f

B.1.6 Retrieval Plans

A retrieval plan p contains the following information:

- **id**: a unique identifier
- **features**: a set of case features
- **problem**: the target (sub)problem description whose features those of p are based on
- **origin**: the origin of p ("user", method, etc.)
- **status**: the status of p: *unexecuted*, *success*, *failure*, etc.
- **actions**: a sequence of retrieval descriptions
- **successful retrieval description**: the retrieval description of p resulting in a succeeding query
- **sources retrieved**: a set of plan descriptions for the candidate source plans retrieved using p
- **the source chosen**: the source plan description for the chosen source plan
- **retrieval description search parameters**: (see Section 4.6)

A sample retrieval plan generated by the Retriever as described in Section 4.6.4, is *RPLAN-1*:

```
Rtvl. Plan RPLAN-1 (RPLAN)
    Derived-via: :AUTO
        Problem: PDESC-3
       Features: (FEATURE-3.1)
         RDESCs: (RDESC-1 RDESC-2)
         Status: :SUCCESS
    Succ. RDESC: RDESC-2
   Sources Rtvd: (SPLAN-1 SPLAN-2)
   Srcs. Chosen: (SPLAN-1)
```

This retrieval plan contains two actions, RDESC-1 and RDESC-2, the first of which failed and the second of which succeeded in the retrieval of two candidate source plans, SPLAN-1 and SPLAN-2.

B.1.7 Plan Descriptions

Source Plan Descriptions

A source plan descriptor is a plan descriptor d that describes a source plan p retrieved by the Plan Retriever. It contains the following information:

- **id**: a unique identifier
- **status**: the status of p: *selected*, *chosen*, etc.
- **frame**: plan frame in the KB for p
- **features**: case features of p that have been instantiated
- **feature mappings**: a set of ⟨target feature,source feature⟩
- **retrieval plan used**: the retrieval plan used to retrieve p
- **retrieval description used**: the retrieval description used to retrieve p
- **score**: similarity metric score of p
- **ranking**: ranking of p by similarity metric score among other candidate sources
- **plan variables**: a list of plan variables in p
- **plan variable constraints**: a list of plan variable constraints in p
- **plan variable mapping information**: (various)

For example, the plan description for the source plan *Plan-Located-At-1.2*, one of the plans retrieved for Case-3 (see Section 4.2), has the following information:

```
SOURCE Plan SPLAN-1
   Frame: Plan-Located-At-1.2
   Successful: True
   Features: (FEATURE-3.1S)
   Mappings: ((FEATURE-3.1 . FEATURE-3.1S))
       Status: :SELECTED
   RPLAN-used: RPLAN-1
   RDESC-used: RDESC-2
        Score: 2
      Ranking: 1
```

B.2 Adaptation

B.2.1 Plan Modification Tasks

A plan modification task descriptor is a working data structure that describes a fault of target plan that needs fixing. It is created by the Plan Checker and has the following information associated with it:

- **id**: a unique identifier
- **plan**: the target plan p_t involved
- **prob**: the target problem which p_t is to solve
- **ptype**: the primary task type
- **stype**: the secondary task type
- **node**: the target plan or e-action involved
- **validation**: the validation involved
- **condition**: the plan/action condition involved
- **parent**: the parent plan modification task
- **prty**: the priority of the task
- **status**: the status of the task
- **modifications**: the modifications made to perform the task

Appendix C

Details of Examples

C.1 Case Case-Make-Deliveries-1

C.2 Plan Plan-CE-Located-At-1.2

C.2.1 Plan-CE-Located-At-1.2 E-Actions

The following are the e-actions, in order, for *Plan-CE-Located-At-1.2*. Actions names are shown followed by their type (action concept Section 3.4.5) in parentheses and roles beneath. Action roles are shown as “role-name: role-filler (plan-variable-value)”.

```
 1. Do-Nothing-1.1 (type: Do-Nothing)
 2. Do-Nothing-1.2 (type: Do-Nothing)
 3. P-Collect-Fees-1.1 (type: P-Collect-Fees)
       package: PVAR-P1 (Pkg-1.1)
 4. Do-Nothing-1.3 (type: Do-Nothing)
 5. P-Open-Door-1.1 (type: P-Open-Door)
       vehicle: PVAR-T1 (Truck-1)
 6. P-Load-Package-1.1 (type: P-Load-Package)

     package: PVAR-P1 (Pkg-1.1)
       vehicle: PVAR-T1 (Truck-1)
 7. P-Close-Door-1.1 (type: P-Close-Door)
       vehicle: PVAR-T1 (Truck-1)
 8. P-Move-Vehicle-1.1
       vehicle:     PVAR-T1   (Truck-1)
       origin:      PVAR-L11C (City1-Cl1)
       destination: PVAR-L12B (City1-Cl2)
       route:       P-Local-Road-Route
 9. P-Open-Door-1.2 (type: P-Open-Door)
       vehicle:     PVAR-T1   (Truck-1)
10. P-Unload-Package-1.1 (type: P-Unload-Package)
       package: PVAR-P1 (Pkg-1.1)
       vehicle: PVAR-T1 (Truck-1)
11. P-Close-Door-1.2  (type: P-Close-Door)
```

```
      vehicle:     PVAR-T1   (Truck-1)
12. Do-Nothing-1.4 (type: Do-Nothing)
13. P-Deliver-1.1 (type: P-Deliver)
      package: PVAR-P1 (Pkg-1.1)
```

C.2.2 Plan-CE-Located-At-1.2 HTN

The following is a listing of the HTN for *Plan-CE-Located-At-1.2*. A graphical representation of the part of the HTN is shown in Figure 3.8.

Plan-CE-Located-At-1.2 HTN By E-Plan

```
Plan-CE-Located-At-1.2:
     Parent: NIL
   Subplans: Plan-CE-Transported-1.1
    Actions: NIL

Plan-CE-Transported-1.1:
     Parent: Plan-CE-Located-At-1.2
   Subplans: Plan-CE-Picked-Up-1.1 Plan-CE-Carried-1.1 Plan-CE-Delivered-1.1
    Actions: NIL

Plan-CE-Picked-Up-1.1:
     Parent: Plan-CE-Transported-1.1
   Subplans: Plan-CE-Pkg-Fees-Collected-1.1 Plan-CE-Pkg-Insured-1.1
             Plan-CE-Have-Pkg-Permit-1.1
    Actions: NIL

Plan-CE-Carried-1.1:
     Parent: Plan-CE-Transported-1.1
   Subplans: Plan-CE-Carried-Direct-1.1
    Actions: NIL

Plan-CE-Delivered-1.1:
     Parent: Plan-CE-Transported-1.1
    Actions: P-Deliver-1.1

Plan-CE-Pkg-Fees-Collected-1.1:
     Parent: Plan-CE-Picked-Up-1.1:
    Actions: P-Collect-Fees-1.1

Plan-CE-Pkg-Insured-1.1:
     Parent: Plan-CE-Picked-Up-1.1:
    Actions: Do-Nothing-1.2

Plan-CE-Have-Pkg-Permit-1.1:
     Parent: Plan-CE-Picked-Up-1.1:
    Actions: Do-Nothing-1.1

Plan-CE-Carried-Direct-1.1:
```

```
     Parent: Plan-CE-Carried-1.1:
   Subplans: Plan-CE-Package-Loaded-Top-1.1 Plan-CE-Located-At-1.1

Plan-Not-CE-Package-Loaded-Top-1.1
    Actions: NIL

Plan-CE-Package-Loaded-Top-1.1:
     Parent: Plan-CE-Carried-Direct-1.1
   Subplans: Plan-CE-Package-Loaded-Haz-1.1 Plan-CE-Package-Loaded-Val-1.1
    Actions: NIL

Plan-CE-Located-At-1.1:
     Parent: Plan-CE-Carried-Direct-1.1
   Subplans: Plan-CE-Vehicle-Moved-1.1
    Actions: NIL

Plan-Not-CE-Package-Loaded-Top-1.1:
     Parent: Plan-CE-Carried-Direct-1.1
   Subplans: Plan-Not-CE-Package-Loaded-Val-1.1
             Plan-Not-CE-Package-Loaded-Haz-1.1
    Actions: NIL

Plan-CE-Package-Loaded-Haz-1.1:
     Parent: Plan-CE-Package-Loaded-Top-1.1:
    Actions: Do-Nothing-1.3

Plan-CE-Package-Loaded-Val-1.1:
     Parent: Plan-CE-Package-Loaded-Top-1.1:
   Subplans: Plan-CE-Package-Loaded-1.2
    Actions: NIL

Plan-CE-Vehicle-Moved-1.1:
     Parent: Plan-CE-Located-At-1.1:
    Actions: P-Move-Vehicle-1.1

Plan-Not-CE-Package-Loaded-Val-1.1:
     Parent: Plan-Not-CE-Package-Loaded-Top-1.1:
   Subplans: Plan-Not-CE-Package-Loaded-1.2
    Actions: NIL

Plan-Not-CE-Package-Loaded-Haz-1.1:
     Parent: Plan-Not-CE-Package-Loaded-Top-1.1:
    Actions: Do-Nothing-1.4

Plan-CE-Package-Loaded-1.2:
     Parent: Plan-CE-Package-Loaded-Val-1.1:
   Subplans: Plan-CE-Door-Open-1.1 Plan-CE-Package-Loaded-1.1
             Plan-Not-CE-Door-Open-1.1
    Actions: NIL

Plan-Not-CE-Package-Loaded-1.2:
```

```
     Parent: Plan-Not-CE-Package-Loaded-Val-1.1:
   Subplans: Plan-CE-Door-Open-1.2 Plan-Not-CE-Package-Loaded-1.1
             Plan-Not-CE-Door-Open-1.2
    Actions: NIL

Plan-CE-Door-Open-1.1:
     Parent: Plan-CE-Package-Loaded-1.2
    Actions: P-Open-Door-1.1

Plan-CE-Package-Loaded-1.1:
     Parent: Plan-CE-Package-Loaded-1.2
    Actions: P-Load-Package-1.1

Plan-Not-CE-Door-Open-1.1:
     Parent: Plan-CE-Package-Loaded-1.2
    Actions: P-Close-Door-1.1

Plan-CE-Door-Open-1.2:
     Parent: Plan-Not-CE-Package-Loaded-1.2
    Actions: P-Open-Door-1.2

Plan-Not-CE-Package-Loaded-1.1:
     Parent: Plan-Not-CE-Package-Loaded-1.2
    Actions: P-Unload-Package-1.1

Plan-Not-CE-Door-Open-1.2:
     Parent: Plan-Not-CE-Package-Loaded-1.2
    Actions: P-Close-Door-1.2
```

Plan-CE-Located-At-1.2 HTN By E-Action

```
Do-Nothing-1.1:
     Parent: Plan-CE-Have-Pkg-Permit-1.1

Do-Nothing-1.2:
     Parent: Plan-CE-Pkg-Insured-1.1

P-Collect-Fees-1.1:
     Parent: Plan-CE-Pkg-Fees-Collected-1.1

Do-Nothing-1.3:
     Parent: Plan-CE-Package-Loaded-Haz-1.1:

P-Open-Door-1.1:
     Parent: Plan-CE-Door-Open-1.1:

P-Load-Package-1.1:
     Parent: Plan-CE-Package-Loaded-1.1:

P-Close-Door-1.1:
     Parent: Plan-Not-CE-Door-Open-1.1:
```

```
P-Move-Vehicle-1.1:
     Parent: Plan-CE-Vehicle-Moved-1.1

P-Open-Door-1.2:
     Parent: Plan-CE-Door-Open-1.2:

P-Unload-Package-1.2:
     Parent: Plan-Not-CE-Package-Loaded-1.1

P-Close-Door-1.2:
     Parent: Plan-Not-CE-Door-Open-1.2:

Do-Nothing-1.4:
     Parent: Plan-Not-CE-Package-Loaded-Haz-1.1:

P-Deliver-1.1
     Parent: Plan-CE-Delivered-1.1
```

C.2.3 Plan Plan-CE-Located-At-1.2 Validations

The following are descriptions of all the validations of plan *Plan-CE-Located-At-1.2*. The condition for each validation is also shown.

```
Plan-Vdtn-1.1:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Plan-CE-Carried-Direct-1.1
Plan-Vdtn-1.1-Cond:
  thing=            PVAR-T1
  type=             PVAR-RT
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.2:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Load-Package-1.1
Plan-Vdtn-1.2-Cond:
  thing=            PVAR-T1
  location=         PVAR-L11A
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.3:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Move-Vehicle-1.1
Plan-Vdtn-1.3-Cond:
  thing=            PVAR-T1
  location=         PVAR-L11C
```

```
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.4:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Plan-CE-Carried-Direct-1.1
Plan-Vdtn-1.4-Cond:
  location=         PVAR-L12B
  city=             PVAR-C1B
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-In-City

Plan-Vdtn-1.5:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Do-Nothing-1.4
Plan-Vdtn-1.5-Cond:
  thing=            PVAR-P1
  type=             Hazardous-Package
  modality=         False
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.6:
  vdtn-Contributor= P-Open-Door-1.2
  vdtn-Consumer=    P-Close-Door-1.2
Plan-Vdtn-1.6-Cond:
  vehicle=          PVAR-T1
  modality=         True
  cond-Type=        Precondition
  instanceOf        CE-Door-Open

Plan-Vdtn-1.7:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Plan-CE-Carried-Direct-1.1
Plan-Vdtn-1.7-Cond:
  thing=            PVAR-P1
  type=             PVAR-RP
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.8:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Plan-CE-Carried-Direct-1.1
Plan-Vdtn-1.8-Cond:
  location=         PVAR-L11B
  city=             PVAR-C1B
  modality=         True
  cond-Type=        Filter-Condition
```

```
  instanceOf        CE-Located-In-City

Plan-Vdtn-1.9:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Move-Vehicle-1.1
Plan-Vdtn-1.9-Cond:
  location=         PVAR-L11C
  city=             PVAR-C1A
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-In-City

Plan-Vdtn-1.10:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Plan-CE-Carried-Direct-1.1
Plan-Vdtn-1.10-Cond:
  thing=            PVAR-T1
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Available

Plan-Vdtn-1.11:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Plan-CE-Carried-Direct-1.1
Plan-Vdtn-1.11-Cond:
  package-Type=     PVAR-RP
  vehicle-Type=     PVAR-RT
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-PV-Compatible

Plan-Vdtn-1.12:
  vdtn-Contributor= P-Load-Package-1.1
  vdtn-Consumer=    P-Close-Door-1.1
Plan-Vdtn-1.12-Cond:
  thing=            PVAR-P1
  location=         PVAR-T1
  modality=         True
  cond-Type=        Precondition
  instanceOf        CE-Located-At

Plan-Vdtn-1.13:
  vdtn-Contributor= Plan-CE-Carried-Direct-1.1
  vdtn-Consumer=    P-Move-Vehicle-1.1
Plan-Vdtn-1.13-Cond:
  thing=            PVAR-T1
  location=         PVAR-L11C
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At
```

```
Plan-Vdtn-1.14:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Do-Nothing-1.3
Plan-Vdtn-1.14-Cond:
  thing=            PVAR-P1
  location=         PVAR-L11B
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.15:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Do-Nothing-1.3
Plan-Vdtn-1.15-Cond:
  thing=            PVAR-P1
  type=             Hazardous-Package
  modality=         False
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.16:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Plan-CE-Carried-Direct-1.1
Plan-Vdtn-1.16-Cond:
  thing=            PVAR-T1
  type=             Truck
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.17:
  vdtn-Contributor= P-Open-Door-1.1
  vdtn-Consumer=    P-Close-Door-1.1
Plan-Vdtn-1.17-Cond:
  vehicle=          PVAR-T1
  modality=         True
  cond-Type=        Precondition
  instanceOf        CE-Door-Open

Plan-Vdtn-1.18:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Open-Door-1.1
Plan-Vdtn-1.18-Cond:
  thing=            PVAR-T1
  location=         PVAR-L11B
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.19:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
```

```
  vdtn-Consumer=    Do-Nothing-1.2
Plan-Vdtn-1.19-Cond:
  thing=            PVAR-P1
  type=             Valuable-Package
  modality=         False
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.20:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Load-Package-1.1
Plan-Vdtn-1.20-Cond:
  thing=            PVAR-P1
  location=         PVAR-L11B
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.21:
  vdtn-Contributor= P-Move-Vehicle-1.1
  vdtn-Consumer=    P-Open-Door-1.2
Plan-Vdtn-1.21-Cond:
  thing=            PVAR-T1
  location=         PVAR-L12B
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.22:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Open-Door-1.1
Plan-Vdtn-1.22-Cond:
  thing=            PVAR-T1
  type=             Regular-Truck
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.23:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Open-Door-1.1
Plan-Vdtn-1.23-Cond:
  thing=            PVAR-P1
  location=         PVAR-L11B
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.24:
  vdtn-Contributor= Plan-CE-Carried-Direct-1.1
  vdtn-Consumer=    Do-Nothing-1.3
```

```
Plan-Vdtn-1.24-Cond:
  thing=            PVAR-T1
  location=         PVAR-L11B
  modality=         True
  cond-Type=        Precondition
  instanceOf        CE-Located-At

Plan-Vdtn-1.25:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Open-Door-1.1
Plan-Vdtn-1.25-Cond:
  thing=            PVAR-P1
  type=             Valuable-Package
  modality=         False
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.26:
  vdtn-Contributor= Plan-CE-Carried-Direct-1.1
  vdtn-Consumer=    P-Open-Door-1.1
Plan-Vdtn-1.26-Cond:
  thing=            PVAR-T1
  location=         PVAR-L11B
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.27:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Move-Vehicle-1.1
Plan-Vdtn-1.27-Cond:
  location=         PVAR-L12B
  city=             PVAR-C1A
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-In-City

Plan-Vdtn-1.28:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Open-Door-1.2
Plan-Vdtn-1.28-Cond:
  thing=            PVAR-T1
  type=             Regular-Truck
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.29:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Move-Vehicle-1.1
Plan-Vdtn-1.29-Cond:
```

```
  thing=            PVAR-T1
  type=             Truck
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.30:
  vdtn-Contributor= P-Move-Vehicle-1.1
  vdtn-Consumer=    P-Unload-Package-1.1
Plan-Vdtn-1.30-Cond:
  thing=            PVAR-T1
  location=         PVAR-L12A
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.31:
  vdtn-Contributor= P-Close-Door-1.1
  vdtn-Consumer=    P-Open-Door-1.2
Plan-Vdtn-1.31-Cond:
  thing=            PVAR-P1
  location=         PVAR-T1
  modality=         True
  cond-Type=        Precondition
  instanceOf        CE-Located-At

Plan-Vdtn-1.32:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    P-Open-Door-1.2
Plan-Vdtn-1.32-Cond:
  thing=            PVAR-P1
  type=             Valuable-Package
  modality=         False
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.33:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Plan-CE-Carried-Direct-1.1
Plan-Vdtn-1.33-Cond:
  thing=            PVAR-T1
  location=         PVAR-L11B
  modality=         True
  cond-Type=        Phantom-Condition
  instanceOf        CE-Located-At

Plan-Vdtn-1.34:
  vdtn-Contributor= Case-Make-Deliveries-1-InitSit
  vdtn-Consumer=    Do-Nothing-1.1
Plan-Vdtn-1.34-Cond:
  thing=            PVAR-P1
```

```
  type=             Hazardous-Package
  modality=         False
  cond-Type=        Filter-Condition
  instanceOf        CE-Has-Type

Plan-Vdtn-1.35:
  vdtn-Contributor= Plan-CE-Carried-Direct-1.1
  vdtn-Consumer=    P-Load-Package-1.1
Plan-Vdtn-1.35-Cond:
  thing=            PVAR-T1
  location=         PVAR-L11A
  modality=         True
  cond-Type=        Filter-Condition
  instanceOf        CE-Located-At
```

Plan-CE-Located-At-1.2 E-Action Validations

The following are all validations for each action of plan *Plan-CE-Located-At-1.2*. The numbers given are the suffixes of the validations described in detail in the previous section. For example, action Do-Nothing-1 has e-precondition 34 which is *Plan-Vdtn-1.34*.

```
Do-Nothing-1 Validations:
   E-Conditions: NIL
E-Preconditions: 34
   P-Conditions: 1 2 3 4 5 7 8 9 10 11 13 14 15 16 18 19 20 22 23 25 26 27
                 28 29 32 33 35
Initial E-Conds: 34

Do-Nothing-2 Validations:
   E-Conditions: NIL
E-Preconditions: 19
   P-Conditions: 1 2 3 4 5 7 8 9 10 11 13 14 15 16 18 20 22 23 25 26 27 28
                 29 32 33 34 35
Initial E-Conds: 19

P-Collect-Fees-1 Validations:
   E-Conditions: NIL
E-Preconditions: NIL
   P-Conditions: 1 2 3 4 5 7 8 9 10 11 13 14 15 16 18 19 20 22 23 25 26 27
                 28 29 32 33 34 35
%%%% FIX NUMBERS BELOW: LEX -> REG
Do-Nothing-3 Validations:
   E-Conditions: NIL
E-Preconditions: 14 15 24
   P-Conditions: 1 10 11 13 16 18 2 20 22 23 25 26 27 28 29 3 32 33 35 4 5
                 7 8 9
Initial E-Conds: 14 15

P-Open-Door-1 Validations:
   E-Conditions: 17
```

```
E-Preconditions: 18 22 23 25 26
   P-Conditions: 1 10 11 13 16 2 20 27 28 29 3 32 33 35 4 5 7 8 9
Initial E-Conds: 18 22 23 25

P-Load-Package-1 Validations:
   E-Conditions: 12
E-Preconditions: 2 20 35
   P-Conditions: 1 10 11 13 16 17 27 28 29 3 32 33 4 5 7 8 9
Initial E-Conds: 2 20

P-Close-Door-1 Validations:
   E-Conditions: 31
E-Preconditions: 12 17
   P-Conditions: 1 10 11 13 16 27 28 29 3 32 33 4 5 7 8 9

P-Move-Vehicle-1 Validations:
   E-Conditions: 21 30
E-Preconditions: 13 27 29 3 9
   P-Conditions: 1 10 11 16 28 31 32 33 4 5 7 8
Initial E-Conds: 27 29 3 9

P-Open-Door-2 Validations:
   E-Conditions: 6
E-Preconditions: 21 28 31 32
   P-Conditions: 1 10 11 16 30 33 4 5 7 8
Initial E-Conds: 28 32

P-Unload-Package-1 Validations:
   E-Conditions: NIL
E-Preconditions: 30
   P-Conditions: 1 10 11 16 33 4 5 6 7 8

P-Close-Door-2 Validations:
   E-Conditions: NIL
E-Preconditions: 6
   P-Conditions: 1 10 11 16 33 4 5 7 8

Do-Nothing-4 Validations:
   E-Conditions: NIL
E-Preconditions: 5
   P-Conditions: 1 10 11 16 33 4 7 8
Initial E-Conds: 5

P-Deliver-1 Validations:
   E-Conditions: NIL
E-Preconditions: NIL
   P-Conditions: NIL
```

Plan-CE-Located-At-1.2 E-Plan Validations

The following are all validations for each subplan of plan *Plan-CE-Located-At-1.2*. The numbers given are the suffixes of the validations described in detail above.

```
Plan-CE-Located-At-1.2 Validations:
   E-Conditions: NIL
E-Preconditions: 1 10 11 14 15 16 18 19 2 20 22 23 25 27 28 29 3 32 33 34 4 5
                 7 8 9
   P-Conditions: NIL
Initial E-Conds: 1 10 11 14 15 16 18 19 2 20 22 23 25 27 28 29 3 32 33 34 4 5
                 7 8 9

Plan-CE-Transported-1 Validations:
   E-Conditions: NIL
E-Preconditions: 1 10 11 14 15 16 18 19 2 20 22 23 25 27 28 29 3 32 33 34 4 5
                 7 8 9
   P-Conditions: NIL
Initial E-Conds: 1 10 11 14 15 16 18 19 2 20 22 23 25 27 28 29 3 32 33 34 4 5
                 7 8 9

Plan-CE-Picked-Up-1 Validations:
   E-Conditions: NIL
E-Preconditions: 19 34
   P-Conditions: 1 10 11 13 14 15 16 18 2 20 22 23 25 26 27 28 29 3 32 33 35
                 4 5 7 8 9
Initial E-Conds: 19 34

Plan-CE-Carried-1 Validations:
   E-Conditions: NIL
E-Preconditions: 1 10 11 14 15 16 18 2 20 22 23 25 27 28 29 3 32 33 4 5 7 8 9
   P-Conditions: NIL
Initial E-Conds: 1 10 11 14 15 16 18 2 20 22 23 25 27 28 29 3 32 33 4 5 7 8 9

Plan-CE-Delivered-1 Validations:
   E-Conditions: NIL
E-Preconditions: NIL
   P-Conditions: NIL

Plan-CE-Pkg-Fees-Collected-1 Validations:
   E-Conditions: NIL
E-Preconditions: NIL
   P-Conditions: 1 10 11 14 15 16 18 2 20 22 23 25 27 28 29 3 32 33 4 5 7 8 9

Plan-CE-Pkg-Insured-1 Validations:
   E-Conditions: NIL
E-Preconditions: 19
   P-Conditions: 1 10 11 14 15 16 18 2 20 22 23 25 27 28 29 3 32 33 4 5 7 8 9
Initial E-Conds: 19

Plan-CE-Have-Pkg-Permit-1 Validations:
```

```
   E-Conditions: NIL
E-Preconditions: 34
   P-Conditions: 1 10 11 14 15 16 18 19 2 20 22 23 25 27 28 29 3 32 33 4 5 7
                 8 9
Initial E-Conds: 34

Plan-CE-Carried-Direct-1 Validations:
   E-Conditions: 13 18 2 24 26 3 35
E-Preconditions: 1 10 11 14 15 16 18 2 20 22 23 25 27 28 29 3 32 33 4 5 7 8 9
   P-Conditions: 13 14 15 18 2 20 22 23 25 26 27 28 29 3 32 35 5 9
Initial E-Conds: 1 10 11 14 15 16 18 2 20 22 23 25 27 28 29 3 32 33 4 5 7 8 9

Plan-CE-Package-Loaded-Top-1 Validations:
   E-Conditions: 31
E-Preconditions: 14 15 18 2 20 22 23 24 25 26 35
   P-Conditions: 1 10 11 13 16 27 28 29 3 32 33 4 5 7 8 9
Initial E-Conds: 14 15 18 2 20 22 23 25

Plan-CE-Located-At-1 Validations:
   E-Conditions: 21 30
E-Preconditions: 13 27 29 3 9
   P-Conditions: 1 10 11 16 28 31 32 33 4 5 7 8
Initial E-Conds: 27 29 3 9

Plan-Not-CE-Package-Loaded-Top-1 Validations:
   E-Conditions: NIL
E-Preconditions: 21 28 30 31 32 5
   P-Conditions: 1 10 11 16 33 4 7 8
Initial E-Conds: 28 32 5

Plan-CE-Package-Loaded-Haz-1 Validations:
   E-Conditions: NIL
E-Preconditions: 14 15 24
   P-Conditions: 1 10 11 13 16 18 2 20 22 23 25 26 27 28 29 3 32 33 35 4 5 7 8
                 9
Initial E-Conds: 14 15

Plan-CE-Package-Loaded-Val-1 Validations:
   E-Conditions: 31
E-Preconditions: 18 2 20 22 23 25 26 35
   P-Conditions: 1 10 11 13 16 27 28 29 3 32 33 4 5 7 8 9
Initial E-Conds: 18 2 20 22 23 25

Plan-CE-Vehicle-Moved-1 Validations:
   E-Conditions: 21 30
E-Preconditions: 13 27 29 3 9
   P-Conditions: 1 10 11 16 28 31 32 33 4 5 7 8
Initial E-Conds: 27 29 3 9

Plan-Not-CE-Package-Loaded-Val-1 Validations:
   E-Conditions: NIL
```

```
E-Preconditions: 21 28 30 31 32
   P-Conditions: 1 10 11 16 33 4 5 7 8
Initial E-Conds: 28 32

Plan-Not-CE-Package-Loaded-Haz-1 Validations:
   E-Conditions: NIL
E-Preconditions: 5
   P-Conditions: 1 10 11 16 33 4 7 8
Initial E-Conds: 5

Plan-CE-Package-Loaded-2 Validations:
   E-Conditions: 31
E-Preconditions: 18 2 20 22 23 25 26 35
   P-Conditions: 10 11 13 16 27 28 29 3 32 33 4 5 7 8 9
Initial E-Conds: 18 2 20 22 23 25

Plan-Not-CE-Package-Loaded-2 Validations:
   E-Conditions: NIL
E-Preconditions: 21 28 30 31 32
   P-Conditions: 1 10 11 16 33 4 5 7 8
Initial E-Conds: 28 32

Plan-CE-Door-Open-1 Validations:
   E-Conditions: 17
E-Preconditions: 18 22 23 25 26
   P-Conditions: 1 10 11 13 16 2 20 27 28 29 3 32 33 35 4 5 7 8 9
Initial E-Conds: 18 22 23 25

Plan-CE-Package-Loaded-1 Validations:
   E-Conditions: 12
E-Preconditions: 2 20 35
   P-Conditions: 1 10 11 13 16 17 27 28 29 3 32 33 4 5 7 8 9
Initial E-Conds: 2 20

Plan-Not-CE-Door-Open-1 Validations:
   E-Conditions: 31
E-Preconditions: 12 17
   P-Conditions: 1 10 11 13 16 27 28 29 3 32 33 4 5 7 8 9

Plan-CE-Door-Open-2 Validations:
   E-Conditions: 6
E-Preconditions: 21 28 31 32
   P-Conditions: 1 10 11 16 30 33 4 5 7 8
Initial E-Conds: 28 32

Plan-Not-CE-Package-Loaded-1 Validations:
   E-Conditions: NIL
E-Preconditions: 30
   P-Conditions: 1 10 11 16 33 4 5 6 7 8

Plan-Not-CE-Door-Open-2 Validations:
```

```
   E-Conditions: NIL
E-Preconditions: 6
   P-Conditions: 1 10 11 16 33 4 5 7 8
```

Validations By Plan Variable

The following are the validations for each plan variable of *Plan-CE-Located-At-1.2*. These validations act as plan variable constraints. The original value of the plan variable is shown in parentheses after its name. The numbers given are the suffixes of the validations described in detail above.

```
PVAR-C1A (City1): 27 9
PVAR-C1B (City1): 4 8
PVAR-L11A (City1-Cl1): 2
PVAR-L11B (City1-Cl1): 14 18 20 23 33 8
PVAR-L11C (City1-Cl1): 3 9
PVAR-L12B (City1-Cl2): 27 4
PVAR-P1 (Pkg-1): 14 15 19 20 23 25 32 34 5 7
PVAR-RP (Regular-Package): 11 7
PVAR-RT (Regular-Truck): 1 11
PVAR-T1 (Truck-1): 1 10 16 18 2 22 28 29 3 33
```

Appendix D

Acknowledgements

My research advisor and thesis committee chairperson, Prof. James Hendler, has given me his advice, support, and encouragement over the past five years. In addition to his suggestions on this work, I've learned a lot from him about how to work in Academia.

Prof. Lindley Darden has been my "antidote" to those spells when I was weary of computer science. My philosophical conversations and collaboration with her on analogical reasoning and the philosophy of molecular biology were a welcome diversion and have broadened my horizons. She also provided detailed comments on drafts of this document.

Prof. Lee Spector has provided many helpful comments on a draft of this document. Thanks to him and my other committee members, Prof. Dana Nau and Prof. Bonnie Dorr, for their participation in the defense and feedback on the dissertation. Dr. Kate Sanders carefully read and commented on my many thesis drafts, sharing her expertise in case-based reasoning and her advice on how to survive the whole thesis experience.

My system, CAPER, draws in part from work of other past and present members of our lab, whom I gratefully acknowledge for their work and also their helpful comments and suggestions on my own work. Bill Andersen produced the bulk of the current version of the PARKA system, which I have made use of in my work. His prolific programming and willingness to fix the bugs I've found are much appreciated. Much credit is due to PARKA's inventors, Prof. Matthew Evett and Prof. Lee Spector, and its user-interface guru, Sean Luke. Prof. Subbarao Kambhampati's work on the PRIAR system has contributed much to my work. Kilian Stoffel collected several timings for CAPER queries on the SP2. Scott Andrews coded a version of the problem generator for CAPER. He and Kutluhan Erol contributed to the design and encoding of the UM Transport Logistics domain. Thanks also to Kutluhan for sharing his wisdom on planning and for his and Bill Regli's participation in our mutual commiseration "ABD Club".

The PLUS (Parallel Understanding Systems) group is a nice place to be for one's graduate experience. Aside from the intellectual stimulation of the many diverse projects going on, I've greatly enjoyed interacting with my fellow graduate students in the group. When I started in the group, I learned a lot about how to be a grad student from the wisdom of the more senior members including Matt Evett, Bill Regli, Lee Spector, and Clare Voss.

My contemporaries in the PLUS group — Bill Andersen, Kutluhan Erol, and Bob Kohout

— have been a big source of moral support and interesting, lively conversations, and friendly debates encompassing such diverse topics as history, politics, economics, the workings of the human mind (answer: interesting and informative), whether A.I. will succeed (answer: hopefully), whether there are any academic jobs to be had in A.I. (answer: nope), and the role of formalism in all of it (answer: hopefully a minimal one).

Since I joined the group, we've amassed more folks who've continued to make this a group a great one. They've included Scott Andrews, Bob Best, Carlos Lopes, Sean Luke, Vikram Manikonda, Barrie Migdalof, Dave Musliner, Dave Rager, Kate Sanders, Oliver Seeliger, Kilian Stoffel, Merwyn Taylor, and Reiko Tsuneto. I wish all my fellow PLUSers the best and hope we can stay in touch!

My thesis work was undoubtedly facilitated by the administrative magic worked by our group secretaries, Edna Walker and Nancy Parker, and by Dr. Nancy Lindley, our graduate program administrator who has all the answers and makes a large department seem a little smaller and more friendly.

My path to graduate school was paved by the fine education I received from the faculty of the University of Massachusetts (Amherst) and Concord-Carlisle High School (Concord, Mass.). At U.Mass, I was especially inspired by Prof. Edwina Rissland's and Prof. Wendy Lenhert's A.I. courses, by Prof. Vere Chappel's Philosophy of Thinking Machines course, and by my interaction with Prof. Linda Lockwood, Director of the Honors Program.

I learned much about working in the computer industry from Karl Hakkarainen at DEC and Joe Spano at IBM. Also thanks to Terry Whearley at Brightware, Inc. for giving me my next job (A.I. is not dead yet!).

Finally, a grateful word to the sponsors of this research. This research was supported in part by grants from NSF(IRI-8907890), ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039 and by ARI (MDA-903-92-R-0035, subcontract through Microelectronics and Design, Inc.).

Bibliography

(Agre and Chapman, 1987) Agre, Phillip E. and Chapman, David 1987. Pengi: An implementation of a theory of activity. In *Proceedings of the International Joint Conference on Artificial Intelligence.* 268–272.

(Aha *et al.*, 1991) Aha, David W.; Kibler, Dennis; and Albert, Mark K. 1991. Instance-based learning algorithms. *Machine Learning* 37–66.

(Allen *et al.*, 1990) Allen, James; Hendler, James; and Tate, Austin, editors 1990. *Readings in Planning.* Morgan Kaufmann Publishers, San Mateo, California.

(Alterman, 1988) Alterman, Richard 1988. Adaptive planning. *Cognitive Science* 12:393–421.

(Andersen *et al.*, 1994) Andersen, William A.; Hendler, James A.; Evett, Matthew P.; and Kettler, Brian P. 1994. Massively parallel matching of knowledge structures. In Kitano, Hiroaki and Hendler, James, editors 1994, *Massively Parallel Artificial Intelligence.* AAAI Press/The MIT Press, Menlo Park, California. 52–73.

(Anderson and Pichert, 1978) Anderson, Richard C. and Pichert, James W. 1978. Recall of previously unrecallable information following a shift in perspective. *Journal of Verbal Learning and Verbal Behavior* 17:1–12.

(Andrews *et al.*, 1995) Andrews, Scott; Kettler, Brian; Erol, Kutluhan; and Hendler, James 1995. UM Translog: A planning domain for the development and benchmarking of planning systems. Technical Report CS-TR-3487, University of Maryland at College Park, Dept. of Computer Science.

(Ashley, 1990) Ashley, Kevin D. 1990. *Modelling legal argument: Reasoning with cases and hypotheticals.* The MIT Press, Cambridge, Mass.

(Barsalou, 1988) Barsalou, Lawrence W. 1988. The content and organization of autobiographical memories. In Neisser, Ulric and Winograd, Eugene, editors 1988, *Remembering Reconsidered: Ecological and Traditional Approaches to the Study of Memory.* Cambridge University Press, Cambridge. chapter 8, 193–241.

(Barsalou, 1992) Barsalou, Lawrence W. 1992. *Cognitive Psychology: An Overview for Cognitive Scientists.* Tutorial Essays in Cognitive Science. Lawrence Erlbaum Associates, Hillsdale, New Jersey.

(Belkin and Cool, 1993) Belkin, N.J. and Cool, C. 1993. The concept of information seeking strategies and its use in the design of information retrieval systems. In *Case-Based Reasoning and Information Retrieval: Exploring the Opportunities for Technology Sharing*, Working Notes. AAAI Spring Symposium Series. 1–7.

(Bobrow and Norman, 1977) Bobrow, B.G. and Norman, D. 1977. An overview of KRL: A knowledge representation language. *Cognitive Science* 1:3–46.

(Boshuizen and Schmidt, 1992) Boshuizen, Henry P.A. and Schmidt, Henk G. 1992. On the role of biomedical knowledge in clinical reasoning by experts, intermediates, and novices. *Cognitive Science* 16:153–184.

(Branting, 1990) Branting, L. Karl 1990. Techniques for retrieval of structured cases. In *Case-based Reasoning: Working Notes*. AAAI, AAAI Spring Symposium. 22–26.

(Cohen *et al.*, 1993) Cohen, Gillian; Kiss, George; and LeVoi, Marvin 1993. *Memory: Current Issues*. Open Guides to Psychology. Open University Press, Buckingham, England, 2nd edition.

(Cohen *et al.*, 1994) Cohen, Paul; Dean, Tom; Gil, Yolanda; Ginsberg, Matt; and Hoebel, Lou 1994. Handbook of evaluation for the arpa/rome lab planning initiative. In Burstein, Mark H., editor 1994, *ARPA/Rome Lab Knowledge-based Planning and Scheduling Initiative Workshop Proceedings*, San Mateo, California. Morgan Kaufmann. 519–533.

(Cohen, 1989) Cohen, Gillian 1989. *Memory in the Real World*. Lawrence Erlbaum Associates, Hillsdale, New Jersy.

(Erol, 1995) Erol, Kutluhan 1995. *HTN Planning: Formalization, Analysis, and Implementation*. Ph.D. thesis, University of Maryland at College Park, Dept. of Computer Science.

(Evett *et al.*, 1994) Evett, Matthew P.; Hendler, James A.; and Spector, Lee 1994. Parallel knowledge representation on the Connection Machine. *Journal of Parallel and Distributed Computing* 22:168–184.

(Evett, 1994) Evett, Matthew P. 1994. *PARKA: A System for Massively Parallel Knowledge Representation*. Ph.D. thesis, University of Maryland at College Park, Dept. of Computer Science. CS-TR-3313.

(Falkenhainer *et al.*, 1990) Falkenhainer, Brian; Forbus, Kenneth D.; and Gentner, Dedre 1990. The structure-mapping engine: Algorithm and examples. *Artificial Intelligence* 41:1–63.

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

(Garey and Johnson, 1979) Garey, Michael R. and Johnson, David S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Bell Telephone Laboratories.

(Gentner and Falkenhainer, 1991) Gentner, Dedre and Falkenhainer, Brian 1991. MAC/FAC: A model of similarity-based retrieval. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, Hillsdale, N.J. Erlbaum. 504–509.

(Gentner and Landers, 1985) Gentner, Dedre and Landers, R. 1985. Analogical reminding: a good match is hard to find. In *Proceedings of the International Conference on Cybernetics and Society*, New York. IEEE. 607–613.

(Gentner, 1983) Gentner, Dedre 1983. Structure-mapping: A theoretical framework for analogy. *Cognitive Science* 7:155–170.

(Gentner, 1989) Gentner, Dedre 1989. The mechanisms of analogical learning. In Vosniadou, Stella and Ortony, Andrew, editors 1989, *Similarity and Analogical Reasoning: A Synthesis*. Cambridge University Press, Cambridge. chapter 7, 199–241.

(Ghosh *et al.*, 1992) Ghosh, Subrata; Hendler, James; Kambhampati, Subbarao; and Kettler, Brian 1992. *UM Nonlin — A Common Lisp Implementation of Nonlin: User Manual*. University of Maryland at College Park, Department of Computer Science.

(Goodman, 1989) Goodman, Marc 1989. CBR in battle planning. In *Proceedings of the Case-based Reasoning Workshop (DARPA)*, San Mateo, California. Morgan Kaufmann. 264–269.

(Green, 1969) Green, C. 1969. Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence*. 219–239.

(Hall, 1989) Hall, Rogers P. 1989. Computational approaches to analogical reasoning. *Artificial Intelligence* 39:39–120.

(Hammond, 1986) Hammond, Kristian J. 1986. *Case-based Planning: An Integrated Theory of Planning, Learning, and Memory.* Doctoral disseration, Yale University, Dept. of Computer Science.

(Hammond, 1990) Hammond, Kristian J. 1990. Case-based planning: A framework for planning from experience. *Cognitive Science* 14:384–443.

(Hanks and Weld, 1995) Hanks, Steve and Weld, Daniel S. 1995. A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research* 2:319–360.

(Hesse, 1966) Hesse, Mary B. 1966. *Models and Analogies in Science.* University of Notre Dame Press, Notre Dame, Indiana.

(Hillis, 1985) Hillis, W. Daniel 1985. *The Connection Machine.* The MIT Press, Cambridge, Massachusetts.

(Holyoak and Koh, 1987) Holyoak, Keith J. and Koh, Kyunghee 1987. Surface and structural similarity in analogical transfer. *Memory and Cognition* 15(4):332–340.

(Holyoak and Thagard, 1989) Holyoak, Keith J. and Thagard, Paul R. 1989. Analogical mapping by constraint satisfaction. *Cognitive Science* 295–355.

(Holyoak and Thagard, 1995) Holyoak, Keith J. and Thagard, Paul 1995. *Mental Leaps: Analogy in Creative Thought.* The MIT Press, Cambridge, Mass.

(Holyoak, 1985) Holyoak, Keith J. 1985. The pragmatics of analogical transfer. *The Psychology of Learning and Motivation* 9:59–87.

(Jones, 1990) Jones, K. Sparck 1990. Information retrieval. In Shapiro, S. C., editor 1990, *Encyclopedia of Artificial Intelligence.* John Wiley, New York. 419–421.

(Josephson and Josephson, 1994) Josephson, John R. and Josephson, Susan G., editors 1994. *Abductive Inference: Computation, Philosophy, Technology.* Cambridge University Press, Cambridge.

(Kahn, 1994) Kahn, Charles E. 1994. Clinical trial and evaluation of a prototype case-based system for planning medical imaging work-up strategies. In Aha, David W., editor 1994, *Case-based Reasoning Workshop, Twelfth National Conference on Artificial Intelligence.* AAAI, unpublished. 138–141.

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

(Kambhampati, 1989) Kambhampati, Subbarao 1989. Flexible reuse and modification in hierarchical planning: A validation structure based approach. Ph.D. Thesis/Technical Report CAR-TR-469, University of Maryland at College Park, Center for Automation Research.

(Kettler and Darden, 1993) Kettler, Brian P. and Darden, Lindley 1993. Protein sequencing experiment planning using analogy. In *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology*, Menlo Park, California. AAAI Press. 215–224.

(Kettler and Hendler, 1994) Kettler, Brian P. and Hendler, James A. 1994. Evaluating a case-based planning system. In Aha, David W., editor 1994, *Case-based Reasoning Workshop, Twelfth National Conference on Artificial Intelligence.* AAAI, unpublished. 157–163.

(Kettler *et al.*, 1993) Kettler, Brian P.; Hendler, James A.; Andersen, William A.; and Evett, Matthew P. 1993. Massively parallel support for case-based planning. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence Applications.* IEEE Computer Society Press. 3–11.

(Kettler *et al.*, 1994) Kettler, Brian P.; Hendler, James A.; Andersen, William A.; and Evett, Matthew P. 1994. Massively parallel support for case-based planning. *IEEE Expert* 8–14.

(Kettler *et al.*, 1995) Kettler, Brian; Andersen, William; Hendler, James; and Luke, Sean 1995. Using the Parka parallel knowledge representation system (version 3.1). Technical Report CS-TR-3485, University of Maryland at College Park, Dept. of Computer Science. (UMIACS TR-95-68, ISR TR 95-56).

(Kitano *et al.*, 1992) Kitano, H.; Shibata, A.; Shimazu, H.; Kajihara, J.; and Sato, A. 1992. Building large-scale and corporate-wide case-based systems. In *Proceedings of AAAI-92*, Cambridge, Mass. AAAI, AAAI Press/The MIT Press.

(Koehler, 1994) Koehler, Jana 1994. Avoiding pitfalls in case-based planning. In Hammond, K., editor 1994, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems.* 104–109.

(Kolodner and Thau, 1988) Kolodner, Janet L. and Thau, Robert 1988. Design and implementation of a case memory. Technical Report RL88-1, Thinking Machines Corporation, Cambridge, Mass.

(Kolodner, 1984) Kolodner, Janet L. 1984. *Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model.* Lawrence Erlbaum Associates, Hillsdale, New Jersey.

(Kolodner, 1988) Kolodner, Janet L. 1988. Retrieving events from a case memory: A parallel implementation. In *Proceedings of the Case-based Reasoning Workshop (DARPA)*, San Mateo, California. Morgan Kaufmann Publishers. 233–249.

(Kolodner, 1989) Kolodner, Janet L. 1989. Judging which is the best case for a case-based reasoner. In *Proceedings of the Case-based Reasoning Workshop (DARPA)*, San Mateo, California. Morgan Kaufmann.

(Kolodner, 1993) Kolodner, Janet L. 1993. *Case-Based Reasoning.* Morgan Kaufmann Publishers, San Mateo, California.

(Kumar, 1992) Kumar, Vipin 1992. Algorithms for constraint-satisfaction problems: A survey. *A.I. Magazine* 13(‘):32–44.

(Lehmann, 1992) Lehmann, Fritz 1992. Semantic networks. *Computers and Mathematics with Applications* 23(2–5):1–50.

(Lenat and Guha, 1990) Lenat, D. B. and Guha, R. V. 1990. *Building Large Knowledge Based Systems.* Addison-Wesley Publishing Company, Reading, Mass.

(Levinson, 1991) Levinson, Robert 1991. Pattern associativity and the retrieval of semantic networks. Technical Report UCSC-CRL-91-14, University of California at Santa Cruz, Board of Studies and Information Sciences.

(Linton, 1986) Linton, Marigold 1986. Ways of searching and the contents of memory. In Rubin, David C., editor 1986, *Autobiographical Memory.* Cambridge University Press, Cambridge. chapter 4, 50–65.

(McCartney and Sanders, 1990) McCartney, Robert and Sanders, Kathryn E. 1990. The case for cases: a call for purity in case-based reasoning. In *Case-based Reasoning: Working Notes.* AAAI, AAAI Spring Symposium. 12–16.

(McCartney, 1993) McCartney, Robert 1993. Episodic cases and real-time performance in a case-based planning system. *Expert Systems with Applications* 6:9–22.

(Miyashita and Sycara, 1993) Miyashita, K. and Sycara, K. 1993. Case-based incremental schedule revision. In Fox, M. and Zweben, M., editors 1993, *Knowledge-based Scheduling.* Morgan Kaufmann, San Mateo, California.

(Nadel, 1988) Nadel, Bernard A. 1988. Tree search and arc consistency in constraint satisfaction algorithms. In Kanal, L. and Kumar, V., editors 1988, *Search in Artificial Intelligence.* Springer-Verlag, New York. 287–243.

(Nilsson, 1980) Nilsson, Nils J. 1980. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers, Los Altos, California.

(Norman and Bobrow, 1979) Norman, Donald A. and Bobrow, Daniel G. 1979. Descriptions: An intermediate stage in memory retrieval. *Cognitive Psychology* 11:107–123.

(Ortony, 1979) Ortony, Andrew 1979. Beyond literal similarity. *The Psychological Review* 86(3):161–180.

(Place and Schmidt, 1992) Place, Allen R. and Schmidt, Thomas G. 1992. *User's Guide for SequenceIt!: An Apprenticeship in Protein Sequencing.* Maryland Marine Biotechnology Institute, Center of Marine Biotechnology.

(Quinlan, 1986) Quinlan, J.R. 1986. Induction of decision trees. *Machine Learning* 1:81–106.

(Reiser *et al.*, 1986) Reiser, Brian J.; Black, John B.; and Kalamarides, Peter 1986. Strategic memory search processes. In Rubin, David C., editor 1986, *Autobiographical Memory.* Cambridge University Press, Cambridge. chapter 7, 101–121.

(Riesbeck and Schank, 1989) Riesbeck, Christopher K. and Schank, Roger C. 1989. *Inside Case-Based Reasoning.* Lawrence Erlbaum Associates, Hillsdale, New Jersey.

(Riesbeck, 1986) Riesbeck, Christopher K. 1986. Direct memory access parsing. In Kolodner, J. L. and Riesbeck, C. K., editors 1986, *Experience, memory, and reasoning.* Lawrence Erlbaum, Northvale, NJ.

(Rissland and Skalak, 1991) Rissland, Edwina L. and Skalak, David B. 1991. CABARET: rule interpretation in a hybrid architecture. *International Journal of Man-Machine Studies* 839–887.

(Sanders *et al.*, 1995) Sanders, Kathryn E.; Kettler, Brian P.; and Hendler, James A. 1995. The case for graph-structured representations. In *First International Conference on Case-Based Reasoning*, New York. Springer-Verlag. in press.

(Sanders, 1994) Sanders, Kathryn E. 1994. CHIRON: Planning in an open-textured domain. Ph.D. thesis, cs-94-38, Dept. of Computer Science, Brown University.

(Schank and Abelson, 1977) Schank, R. C. and Abelson, R. P. 1977. *Scripts, Plans, Goals, and Understanding.* Lawrence Erlbaum Associates, Hillsdale, New Jersey.

(Schank and Riesbeck, 1981) Schank, Roger C. and Riesbeck, Christopher K. 1981. *Inside Computer Understanding: Five Programs Plus Miniatures.* Lawrence Erlbaum Associates, Hillsdale, New Jersey.

(Schank, 1982) Schank, Roger C. 1982. *Dynamic Memory: A theory of learning in computers and people.* Cambridge University Press, New York.

(Schank, 1986) Schank, Roger C. 1986. *Explanation Patterns: Understanding Mechanically and Creatively.* Lawrence Erlbaum Associates, Hillsdale, New Jersey.

(Shavlik and Dietterich, 1990) Shavlik, Jude W. and Dietterich, Thomas G. 1990. *Readings in Machine Learning.* Morgan Kaufmann Publishers, San Mateo, California.

(Spector, 1992) Spector, Lee 1992. Supervenience in dynamic-world planning. Technical Report CS-TR-2899, University of Maryland at College Park, Department of Computer Science.

(Stanfill and Waltz, 1986) Stanfill, Craig and Waltz, David 1986. Toward memory-based reasoning. *Communications of the ACM* 29(12):1213–1218.

(Tate, 1977) Tate, Austin 1977. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.* 888–893.

(Thagard and Holyoak, 1989) Thagard, Paul R. and Holyoak, Keith J. 1989. Why indexing is the wrong way to think about analog retrieval. In *Proceedings of the Case-based Reasoning Workshop (DARPA)*, San Mateo, California. Morgan Kaufmann.

(Thagard *et al.*, 1990) Thagard, Paul R.; Holyoak, Keith J.; Nelson, Craig; and Gochfeld, David 1990. Analog retrieval by constraint satisfaction. *Artificial Intelligence* 46:259–310.

(Tulving, 1983) Tulving, E. 1983. *Elements of Episodic Memory.* Oxford University Press, New York.

(Tversky, 1977) Tversky, Amos 1977. Features of similarity. *The Psychological Review* 84(4):327–352.

(Veloso, 1992) Veloso, Manuela M. 1992. *Learning By Analogical Reasoning in General Problem Solving.* Ph.D. Dissertation, Carnegie Mellon University, School of Computer Science.

(Vosniadou and Ortony, 1989) Vosniadou, Stella and Ortony, Andrew, editors 1989. *Similarity and Analogical Reasoning.* Cambridge University Press, Cambridge.

(Waltz, 1989) Waltz, David L. 1989. Is indexing used for analog retrieval? In *Proceedings of the Case-based Reasoning Workshop (DARPA)*, San Mateo, California. Morgan Kaufmann. 41–45.

(Williams and Hollan, 1981) Williams, Michael D. and Hollan, James D. 1981. The process of retrieval from very long-term memory. *Cognitive Science* 5:87–119.

(Winston, 1984) Winston, Patrick Henry 1984. *Artificial Intelligence.* Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition.