Integrating Twofold Case Retrieval and Complete Decision Replay in CAPLAN/CBC

Vom Fachbereich Informatik der Universität Kaiserslautern
zur Verleihung des akademisches Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation von

Héctor Muñoz-Avila

Berichtstatter:	Prof. Dr. Michael M. Richter
	Prof. Dr. Manuela Veloso
Dekan:	Prof. Dr. Michael M. Richter

Datum der wissenschatlichen Aussprache: 18. Mai 1998

Acknowledgements

First, I would like to thank Prof. Dr. Michael M. Richter for the guidance provided over all this time and for admitting me in our research group, which Prof. Richter has made the ideal research environment where hard work is combined with the adequate dosis of liberty. I would also like to thank Prof. Dr. Manuela Veloso for the helpful discussions when I began this research and at the intermediate stage of my work. They oriented my research in a decisive way. I thank Prof. Veloso also for making the long trip from Pittsburgh to Kaiserslautern for my thesis defense. I want to thank Prof. Dr. Klaus Madlener for making it possible for me to study in Kaiserslautern and for the provided council.

I would like to thank my colleagues for their help and support. In particular, I would like to thank: Frank Weberskirch for his invaluable work in the project CAPlan. Without his permanent administration of the CAPlan system, none of this work would have been possible; Jürgen Paulokat for the constructive critic on diverse aspects of my earlier work that contributed to its maduration; Ralph Bergmann for the many discussions on different aspects of case-based planning; Stefan Weß for the early introduction in the CAPlan project; César Carranza and Harald Holz for the discussions about general aspects of my work.

Special thanks to Jochem Hüllen for implementing most components of my work. I would also like to thank Thomas Roth-Berghofer and Werner Schirp for the implementation of the other components.

I thank all members of our research group for their cooperation: Klaus-Dieter Althoff, Barbara Dellen, Christoph Globig, Frank Maurer, Gerd Pews, Reinhard Praeger, Jürgen Rahmel, Aldo von Wangenheim, Oliver Wendel, Stefan Weß, César Carranza, Dirk Krechel, Harald Holz, Sascha Schmitt, Boris Kötting, Sigrid Goldmann, Fawsy Bendeck and Ivo Vollrath. Special thanks to Edith Hüttel-Schwichtenberg, Wolfgang Wilke, Ralph Bergmann, Harald Holz, and Frank Weberskirch for their support.

For the corrections of this work, I would like to thank Frank Weberskirch, Ralph Bergmann and César Carranza. For the final corrections, I would also like to thank Prof. Richter and Prof. Veloso.

I thank my parents and my sister for all of their support and love. Special thanks to I-Hsuan Lin for her support, help and love over all these years.

Kaiserslautern, May 1998,

Héctor Muñoz-Avila

Abstract

In this thesis, different techniques for performing retrieval, adaptation and learning will be presented and integrated in the case-based planner CAPLAN/CBC. The main purpose of this thesis is to improve the performance of the case-based planning process.

Retrieval in CAPLAN/CBC is twofold integrating static and dynamic retrieval techniques. The static retrieval technique is dependency-driven retrieval and is motivated by the necessity to handle some kind of information available in certain complex domains. The dynamic retrieval technique is feature weighting; after each case-based planning episode takes place, the performance of the retrieved cases is evaluated. Elements determining the value of the similarity assessment are changed depending on the result of the evaluation. Dependency-driven retrieval guarantees a certain level of reliability in the retrieval whereas feature weighting dynamically improves it. By combining these two techniques, a powerful retrieval method is obtained.

The adaptation method implemented in this thesis is complete decision replay, an extension of standard replay for plan-space planners. In complete decision replay not only the information leading to the solution is stored in the cases but information about failed attempts is stored as well. This information is considered during the adaptation process of the cases to facilitate their refitting in the current situation. This technique also enables the user to interact during the adaptation process. An integration issue related to complete decision replay is that the system maintains statistical information about the reliability of retrieval. If the reliability of a case is improved as a result of the feature weighting process, standard replay is performed instead of complete decision replay because the case is likely to fit in situations in which it is retrieved.

New cases are created only if during case-based planning episodes the guidance provided by the available cases is considered nonbeneficial. To determine if the guidance is beneficial or not, a measure of the effort made to refit the cases in the current situation is computed. This policy to create new cases is motivated by complete decision replay, which facilitates the refitting process, and by feature weighting, which improves the accuracy of the retrieval.

The advantages and disadvantages of each method are discussed as well as the way they complement each other. The characteristics of the domain for which the different methods are suitable are stated. The results are supported by extensive empirical validation.

Contents

Part I

Introduction

Chapter 1

Introduction

General problem solving is one of the oldest research fields in Artificial Intelligence (AI). Early research shows that problem solving is equivalent to searching a node in a certain graph (Newell and Simon, 1963; Korf, 1987). A major difficulty is that these *search graphs* are usually of exponential size. Thus, developing adequate methods to traverse them is crucial to minimize the time effort to solve a problem (Russell and Norvig, 1996; Richter, 1992). During the search process, the nodes of these graphs become *choice points* in the sense that a decision must be made regarding which of the neighbouring nodes should be visited next.

Planning is an instance of general problem solving in which a sequence of actions must be found to transform the state of the world into a required state (Fikes and Nilsson, 1971; Fikes et al., 1972). Several planning paradigms such as state-space and plan-space planning have been developed to find the sequence of actions. They mainly differ in the way the intermediate states obtained during the planning process are seen; namely, whereas in state-space planning the intermediate states indicate states of the world, in plan-space planning the intermediate states indicate states of the partial solution. Several studies have been made to investigate in which situation which paradigm is better (Barrett and Weld, 1994; Veloso and Blythe, 1994; Kambhampati et al., 1996a).

Different methods have been developed to guide the planning process. They support the planner in the decision-making process at the choice points. Regardless of the underlying planning paradigm used, these methods can be classified as static or dynamic. Static methods are defined previously to the problem solving episodes whereas dynamic methods change either during or after a problem solving episode. In the latter situation, the planner may make different decisions when confronted with the same choice points at different points of time.

Typically, dynamic methods to guide the planning process are obtained by learning from problem solving episodes. Methods that learn from problem solving episodes have been divided into knowledge intensive and lazy learning methods. The first ones are characterized by the fact that they compile the knowledge learned corresponding to the situation found at every choice point and used the compiled knowledge eagerly in latter problem solving episodes (i.e., at every choice point). A typical knowledge intensive method is explanation-based learning (EBL), in which rules to guide the base-level planner are learned (Mitchell et al., 1986; Minton, 1988; Minton et al., 1989; Katukam and Kambhampati, 1994; Kambhampati et al., 1996b). The rules compile situations encountered at the choice points. They also indicate which decision was made and if the decision was successful (i.e., the decision leads the problem solver towards a solution). If, in a problem solving episode, a decision at a choice point must be made and the decision taken is wrong, the planner makes another decision if it is confronted with the same situation in future problem solving episodes.

A well known problem of knowledge intensive methods is the difficulty of handling the compiled knowledge that has been learned (Minton, 1988; Minton, 1990). In particular, the performance gains of EBL are decreased with the number of learned rules. Eventually, after several rules have been learned, the performance of the planning process with EBL decreases and may result worse than planning from scratch. This is called the *utility problem.*

Case-based Reasoning (CBR) overcomes the utility problem by providing efficient access methods to the learned knowledge. In contrast to EBL, learning does not take place after making a decision at every choice point. Instead, part of the planning episode is stored as a whole to be reused in future problem solving episodes. That is, instead of analyzing the situations occuring at every choice point and learning from them, a part of the explored search graph is stored.

In the last years, CBR has been the subject of increasing interest because of several successful applications that show the potentials of this technique (Kolodner, 1993; Aamodt and Plaza, 1994; Althoff et al., 1995; Leake, 1996). CBR has been used in different areas such as in diagnosis, in monitoring, in broadcasting, and of particular interest for this thesis in planning.

1.1 The Need for Case-based Planning

In AI planning, general-purpose architectures have been developed. As with the expert systems, the main motivation of these architectures is to decouple knowledge representation from inference. In this way, the same architecture may be used, for example, to plan a sequence of actions to transport radioactive material between a nuclear plant and a silo or to transform a piece of raw material into a mechanical workpiece.

One of the main difficulties with general-purpose architectures is generating adequate control strategies to plan for the sequence of actions solving a problem. The use of control rules is usually not a feasible choice for two reasons: first, whereas the knowledge about actions transforming the world can be modelled directly by observing processes in the problem domain, the knowledge about how to reason with such actions may be very difficult to acquire; extracting the latter knowledge results in the bottle-neck typical of knowledge engineering. Second, acquiring the control rules by learning from previous problem-solving episodes results in the utility problem.

Case-based planning (CBP) offers an alternative to rules for controlling the general-purpose architectures. In CBP, previously-obtained solution plans and their problem descriptions are stored as cases. Given a new problem, a *similar* case is retrieved and its solution is *adapted* to solve the new problem. *Similarity* and *adaptation* are key aspects determining the performance of case-based planners.

1.2 Issues about Case-based Planning

In general, any case-based planner must address the following issues (Veloso, 1994):

1. Similarity assessment. The similarity assessment must predict if a given case can be easily adapt to solve the new problem (Smyth and Keane, 1994).

2. Organization of the case base. The case base must be structured in a way that enables to evaluate the similarity assessment on the existing cases in an efficient way.

3. Adaptation of cases. Once one or more similar cases to the new problem have been retrieved, their solutions must be adapted to build a solution of the new problem.

Several general-purpose, case-based planners have been developed that assess one or more of these issues (Veloso, 1994; Kambhampati, 1994; Koehler, 1994; Francis and Ram, 1995b; Bergmann and Wilke, 1995a; Ihrig and Kambhampati, 1996a). The contributions of these case-based planners that are relevant to this thesis can be categorized as follows:

1. **Goal-Driven Retrieval.** *Planning problems* are described as a pair (I, G) where I represents the initial conditions or features of the problem and G are the goals to be achieved (Fikes and Nilsson, 1971). Case-based planners traditionally first examine the goals of the candidate cases, preselecting the ones that achieve one or more goals of the new problem. The final selection is made by comparing the initial conditions of the preselected cases. Most of the case libraries known in the literature reflect this principle by indexing the cases by their goals at the top level (Veloso, 1994; Francis and Ram, 1995b; Ihrig and Kambhampati, 1996a).

2. **Static Similarity Metrics.** For synthesis tasks such as planning, similarity metrics predict the adaptation effort of the cases to the new problem. Some similarity metrics take into account relevant features instead of all the features stated in the problem descriptions (Veloso, 1994). Others analyze the contribution of the features to a particular solution and rate them accordingly

(Kambhampati, 1994). But common to all of them is the fact that the similarity metrics are static in that the measure between a case and a problem is always the same.

3. **Relevance of Features.** A case is usually seen as a pair $((I, G), Sol)$, where (I, G) is a problem description and Sol is a solution plan for (I, G). There can be several solution plans for the same problem. It has been observed that the relevance of a feature in I depends on the particular solution Sol (Veloso and Carbonell, 1993); whereas a feature may be relevant for a particular solution, it may not be relevant for another solution. A method, known as the footprinting process, has been developed to identify if a feature is relevant for a particular solution (Veloso, 1994). Base of this method is the goal regression process used in EBL (Mitchell et al., 1986).

4. **Adaptation with Analogical Replay.** Most of the adaptation methods in CBP are based on replay. Under this approach the cases are viewed as *derivational paths* indicating which decisions were taken at the choice points (Veloso and Carbonell, 1993). Initially, the method was implemented on a state-space, case-based planner (Veloso, 1994) but later it has been implemented in case-based, plan-space planners as well (Ihrig and Kambhampati, 1994; Muñoz-Avila et al., 1994). During the adaptation phase the derivational path is reconstructed relative to the conditions of the new problem by taking the same decisions. This paradigm has been extended by developing a language to perform anotations on the derivational path (Veloso, 1994). These anotations indicate failed decisions at the choice points. The language represents situations occuring during state-space planning. For plan-space planners only the replay method has been used but no attempts to express the failures have been made until now.

5. **Trade-Off between Efficiency Gains and Case Search.** Most case-based planners retrieve multiple cases to solve new problems (Veloso, 1994; Kambhampati, 1994; Francis and Ram, 1995b; Ihrig and Kambhampati, 1996a). Each case covers one or more goals of the new problem in pursue of covering as much of the goals as possible. Methods for merging them vary depending on the underlying planner. For state-space planners, merging can be done by interleaving first-principles planning and replay (Veloso, 1994). For plan-space planners, replay is done first and then first-principles planning follows (Ihrig and Kambhampati, 1994). This is based on the capability of plan-space planners to decouple plan step execution from plan step ordering, which allows them to interleave steps in the plan. In general, however, the case-based planner should not espend arbitrary time searching for the cases to be retrieved; given that the search has time costs, there is a point where it doesn't pay-off to further search for more cases (Veloso, 1994).

6. **Policy to Create New Cases based on Retrieval Failure.** Most systems create new cases eagerly; every time a new solution is found, it is stored together with the corresponding problem description as a new case. Others, follow a more elaborated policy: new cases are created only if the retrieved case is found not to fit into a solution plan of the new problem. That is, if parts of the case need to be revised to find a solution plan. In such situations the retrieval is said to be a failure (Ihrig and Kambhampati, 1996a).

An overview of other issues about case-base planning will be made in chapter 10.

1.3 Problems Studied in this Thesis

In this thesis the six issues cathegorized in the previous section were carefully studied. We made this study by examining their performance in complex domains and found several problems that needed to be studied:

1. **Handling Extended Problem Descriptions.** In many complex domains more information about the problem is known than just the initial conditions or features I and the goals to be achieved G. More concrete, in these domains a partial order, $\prec$, can be predetermined that indicates ordering constraints for achieving the goals G. In this context, problems are given in the form of *extended problem descriptions* $(I, G, \prec)$. The problem is whether a retrieval technique can be developed to select cases in these domains.

2. **Determining a Ranking between the Relevant Features of a Case.** In many domains, case features may not only be classified between relevant and nonrelevant, but some relevant features may be more important than other relevant features for a case. The question is how to determine a ranking of the features in the cases according to their importance. If such a rank would be available, the retrieval procedure can be improved by considering the feature ranks.

3. **Determining the Context of a Feature.** Related to the previous issue, in many situations is not only useful to know which features are relevant for a case but to know the context of the case features. That is, the factors affecting the ranking of features within a case. This is of particular importance for case retrieval, as usually no case is available that solves the current problem and, thus, similarity is determined by a partial match between the features. If the feature context is known, the impact of the absence of a case feature in the current problem can be predicted to determine if the case should be retrieved or not.

4. **Adapting Cases in Plan-Space Planning.** Adaptation based on replay has been implemented for plan-space planners. However, these methods consider

only the sequence of decisions that solve the problem stored in the cases. A more powerful adaptation mechanism should also consider failed attempts made when the problem was solved. The problem is how to implement an adaptation method that also considers the failed attempts during the adaptation process based on a plan-space planner.

5. **Considering User Interactions during Case Adaptation.** Current case-based planners are closed in the sense that the user is not considered during the adaptation process. In realistic situations, however, it is likely that the user would like to interact during the adaptation process by indicating, for example, which parts of the retrieved case should not be considered.

6. **Merging Multiple Cases.** Because of their least-commitment strategy (not to force orderings between plan steps unless it is necessary to handle interactions), case-based, plan-space planners have been believed to be particularly suited to merge multiple cases (Ihrig and Kambhampati, 1996b). The question is whether multi-case merging is in fact efficient in general or if there are situations in which multi-case merging is not as effective as it has been claimed.

7. **Creating New Cases.** As mentioned in point 6 in the previous section, new cases are created either eagerly or when a failure occurs. Clearly, an adequate policy should reduce the redundancy of the case base by avoiding the creation of cases which are already covered by the existing cases. The question in this situation is if retrieval failure is an adequate policy for case creation and if not, which case creation policy is effective to reduce the redundancy of the case base.

In this thesis we developed several techniques to overcome these problems. Further, we proposed an unified framework integrating these techniques in a unified framework.

1.4 Contributions of the Thesis

During the development of this thesis, the case-based planner CAPLAN/CBC has been conceived and implemented. CAPLAN/CBC is coupled with the generic, plan-space planner CAPLAN.[1] CAPLAN is an architecture that provides an interface allowing an external agent to control its planning process. The case-based planner CAPLAN/CBC uses this interface to guide CAPLAN by reusing previous problem solving experiences stored as cases. Case Retrieval in CAPLAN/CBC is twofold; it combines static and dynamic retrieval techniques. The retrieval is integrated with a powerful adaptation mechanism to solve new problems. CAPLAN/CBC makes

[1]The name "CAPLAN" denotes computer assisted planning and "CBC" denotes case-based control.

several contributions to CBP, which overcomes each of the problems discussed in the previous section.

1. **Dependency-Driven Retrieval.** Dependency-driven retrieval is the technique developed in CAPlan/CbC to handle problems that are given in the form of extended problem descriptions $(I, G, \prec)$. To take advantage of this information, the dependencies, $<_C$, between the goals achieved in each case C are represented explicitly. The dependencies reflect the order in which the goals are achieved in C and they are represented at the top level of the case base in CAPlan/CbC (Muñoz-Avila and Hüllen, 1995; Muñoz-Avila and Weberskirch, 1996b). During retrieval, the dependencies $<_C$ between the goals in the cases are compared against the ordering restrictions $\prec$ of the problems. This means that retrieval in CAPlan/CbC is driven by the dependencies instead of the goals as in other case-based planners (see Chapter 4).

2. **Dynamic Similarity Metrics.** In CAPlan/CbC features are given a weight. These feature weights represent a hypothesis about the relative relevance of the feature in the case. The similarity metric counts these feature weights during retrieval to determine if the case should be retrieved to solve the current problem (Muñoz-Avila and Hüllen, 1996). Once a case is retrieved, CAPlan/CbC evaluates if the retrieval was adequate. If this is the situation, the hypothesis about the relevance of the features is reinforced. Otherwise, the hypothesis is punished. Reinforcement and punishment of a hypothesis about the relevance of a feature is made by updating its weight. That is, the feature weight is increased or decreased relative to the other feature weights in the case. The updated weights indicate again a new hypothesis about the relevance of the features in the case and this hypothesis is tested in future retrieval episodes (see Chapter 6).

3. **The Context of a Feature.** Determining the context of a feature (i.e., the factors affecting the ranking of features within a case) has been the subject of several studies in analysis tasks such as classification tasks (Aha and Goldstone, 1990; Turney, 1996) but not in synthesis tasks such as planning. We address this problem and determine that the domain theory plays a key role in determining the context of a feature (Muñoz-Avila and Weberskirch, 1997b; Muñoz-Avila et al., 1997). We provide a characterization for the domain theory and show that in domains meeting this characterization, the context can be simplified (see Chapter 6).

4. **Adaptation in Plan-Space Planning with Complete Decision Replay.** One distinguished characteristic of the base-level planner CAPlan is the incorporation of a structure representing dependencies between goals and decisions (Weberskirch, 1995). This enables CAPlan to construct the justifications for every decision made during the planning process. As a result, CAPlan is able

to perform interactive planning; the user may dynamically change conditions of the problem and CAPlan is able to modify the current plan without having to plan from the scratch. CAPlan/CbC takes advantage of this structure by storing in the cases not just the solution trace (i.e., the sequence of decisions that solve the problem) but the whole dependency structure (Muñoz-Avila and Weberskirch, 1996b). To adapt the retrieved cases, their dependency structures are reconstructed relative to the new problem. The result is a powerful adaptation method that enables CAPlan/CbC to:

1. Handle the initial conditions of the current problem in a straigthforward way. By using the dependency structure, the parts of the plan that are not valid because of missing conditions are easily identified and removed.
2. Reconstruct the whole planning episode. If the justifications for a failed decision can be reconstructed, the failed decision is marked as invalid relative to the current problem. As a result during the completion of the partial solution obtained from the cases, the base-level planner CAPlan will not explore the failed decisions.
3. Adapt the case with user interaction. By reconstructing the dependency structure, the functionality of the base-level planer CAPlan is naturally inherited. As a result, the user is able to dynamically prune parts of the retrieved case or state the invalidity of initial conditions during the adaptation process.

The second point is similar to the effect obtained by using the language for the failed attempts in Prodigy/Analogy. However, reconstructing the dependency structure enables CAPlan/CbC to adapt with less effort the cases (point 1), enables the user to interact with the system based on the partial solution obtained from the cases (point 3) and also enables CAPlan to use efficient backtracking methods such as dependency-directed backtracking (see Chapter 5).

5.**Trade-off between Efficiency Gains and Case Merging.** As mentioned before, case-based, plan-space planners have been believed to be particularly suited to merge multiple cases. In this thesis, two forms of merging in the context of plan-space planning will be examined. We will show that similar to the trade-off between efficiency gains and the number of cases examined during retrieval, there is a trade-off between the efficiency gains and the number of cases being merged for domains having a certain form of interactions (Muñoz-Avila and Weberskirch, 1997a). This means that after merging a certain number of cases, it doesn't pay-off to merge more cases because no efficiency-gains will be made (see Chapter 8).

6. **Policy to Create New Cases based on Retrieval Benefits.** In a case-based planning episode, if a retrieval failure occurs, it does not necessarily

means that the adaptation effort was significant. In addition, the retrieval of a case may not have been a failure because the case fits into a solution plan of the problem. However, the effort to extend the case into the solution plan may have been considerable. CAPlan/CbC stores the solution found together with the problem description as a new case only if the adaptation effort is considered significant and independent of the fact that a retrieval failure occured or not (see Chapter 7).

In this thesis, each of these issues will be examined carefully. Advantages and disadvantages of the implemented methods will be pointed out, and characterizations of the domains in which using the methods is an adequate choice will be given. Moreover, we will show how they are integrated in a problem solving framework by the case-based planner CAPlan/CbC (see Chapters 3, 7). The integration aspects included in the framework are:

Organization of the Case Base. By combining static and dynamic retrieval techniques, CAPlan/CbC improves the accuracy of the retrieval provided that cases are found meeting the conditions proper of each technique. This would be worthless if no mechanism is provided allowing to test these conditions in an efficient way. For this reason, an indexing structure is developed that enables CAPlan/CbC to perform the twofold retrieval process by traversing the indexing structure instead of testing the retrieval conditions case by case (see Chapters 4 and 7).

Dual Integration of Retrieval and Adaptation. In CAPlan/CbC, the retrieval phase not only determines which cases will be selected but it selects the adaptation method itself. The dynamic selection of the adaptation method is based on the reliability of the retrieved cases. When nonrelialable cases are retrieved, complete decision replay is selected because this method is specially suited for fixing the cases. The reliability of a case tends to increase as a result of the feature weighting process. When the case is relialable, standard replay can be used to improve the performance of the case-based planning process (see Chapter 7).

1.5 Organization of the Thesis

The thesis is organized in four parts. The first part is an introduction to the thesis. The second part contains the contributions of this thesis. Part 3 discusses related work and presents conclusions of this work and Part 4 contains the appendix. This document is finished with the bibliography, the glossary and the index.

Part 1 is composed of two chapters, one of which is this one. Chapter 2 contains an overview of different existing techniques that are basic to understand the contributions of this thesis.

Part 2, which is the kernel of the thesis, is divided in seven chapters: Chapter 3 presents the problem solving cycle in CAPlan/CbC. The next chapter presents the dependency-driven retrieval technique. The adaptation technique complete decision replay is presented in Chapter 5. Chapter 6 discusses how feature weighting was conceived and developed for case-based planning and how the context of a feature can be determined. Chapter 7 presents a complete overview of CAPlan/CbC and shows how the contributions are integrated in an unified framework. The study on mergeability in the context of plan-space planning is presented in Chapter 8. The claims and results made in this thesis are validated empirically in Chapter 9.

Part 3 consists of two chapters. Chapter 10 presents related work, including other case-based planners and feature weighting in analysis tasks such as diagnosis. The conclusions of this thesis as well as future work are presented in Chapter 11.

Part 4 contains the appendix. Appendix A presents a symbolic description of the domain of process planning. The specification was developed by the author as part of this thesis and the problems in this domain motivated some of the techniques presented here. Appendix B presents the symbolic specification of the logistics transportation domain (Veloso, 1994). Finally, Appendix C presents the symbolic description of an artificial domain.

Chapter 2

Basic Concepts

This chapter introduces some basic concepts about first-principles planning. It does not intent to be comprehensive but rather to present the basic notations and concepts that will be used in the subsequent chapters. A well-written introduction to AI planning can be found in (Russell and Norvig, 1996) (Part IV). In addition, some basic concepts about case-based planning (CBP) will be presented.

2.1 The Symbolic Specification of Domains

In AI planning, a symbolic specification is used to represent the problem domains. Typically, symbolic specifications can be divided into objects, predicates and operators.

Objects. Objects are the basic representational units. They describe the entities in the world. Examples of objects in the domain of process planning to manufacture mechanical workpieces are the processing areas of the workpieces and the cutting tools. Symbolically, an object is represented as a string of characters. For example, H may indicate the processing area "horizontal outline" or *lrt* may indicate the cutting tool "left rotary tool". The base-level planner CAPLAN represents type information about the objects. Thus, it can be represented symbolically that, for example, H is an object of type "horizontal outline".[1]

Predicates. Predicates indicate relations between objects. Examples of a predicate in the domain of process planning are *subarea*(U_1,H) and *processed*(H). The former indicates the relation "the processing area U_1 is a subarea of H" and the latter indicates the relation "H has been processed". The term "predicate" as used here corresponds to what in mathematical logic is called a literal, that is, an atomic ground

[1]Other systems represent type information as well. In CAPLAN, however, this information is used to partially solve the so-called problem of the filter conditions (Weberskirch and Muñoz-Avila, 1997).

predicate where the arguments are objects (i.e., no argument is a function term or a variable). We will, however, use indistinctly the word "predicate" to refer to both (it should be clear from the context to which one we are refering).

Operators. Operators describe actions, which are the basic units from which plans are made. A widely used representation for operators are the STRIPS-operators (Fikes and Nilsson, 1971). These operators are constituted of arguments, constraints, preconditions and effects.

- Arguments. The arguments list all variables used in the definition of the operator.
- Constraints. The constraints are the binding constraints on the variables. There are codesignation constraints, *Same(<var>,<var'>)*, and noncodesignation constraints, *NotSame(<var>,<var'>)*, indicating that the variable *<var>* must (not) be instantiated to the same object as the variable *<var'>*. In addition, *IsOfType(<type>,<var>)* and *IsNotOfType(<type>,<var>)* are type constraints indicating that the binding of the variable *<var>* must (not) be of type *<type>*.
- Preconditions. The preconditions of an operator list the conditions that must hold hold in the world before the operator can be executed. A precondition has the form: *+predicateName(<var-1>, ..., <var-n>)*, where <var-i> is a variable.
- Effects. The effects of an operator list the changes to the world that the application of the action causes. These changes have the form *+predicateName(<var-1>, ..., <var-n>)*, if the effect is in the add-list, and, *-predicateName(<var-1>, ..., <var-n>)*, if the effect is in the delete-list. That is, the effect is to be added or deleted from the current state. *+/−predicateName(<var-1>, ..., <var-n>)* is called a variable predicate.

Figure 2.1 shows the definition of the holding operators *HoldTool* and *MakeToolHolderFree*. The operator *HoldTool* requires the tool holder to be free. The add-list consists of a single effect; namely, that *tool* is held. The delete-list deletes the condition stating that the tool holder is free. The operator *MakeToolHolderFree* reverses *HoldTool*. It requires that a tool is been held. The effects of the operator is that *tool* is no longer held and the tool holder is free.

2.2 Planning Problems and Solution Plans

In AI planning, the definition of a problem description is based on the concept of *state* of the world. An state can be seen as a snapshot of the world and is represented as a (finite) set of predicates. That is, a state describe the relations between the objects of the world that hold at a certain time.

Operator: *HoldTool*	**Operator:** *MakeToolHolderFree*
Arguments:	**Arguments:**
tool	tool
Constraints:	**Constraints:**
IsOfType(Tool,tool)	IsOfType(Tool,tool)
Effects:	**Effects:**
+toolHeld(tool)	+toolHolderFree()
−toolHolderFree()	−toolHeld(tool)
Preconditions:	**Preconditions:**
+toolHolderFree()	+toolHeld(tool)

Figure 2.1: Definition of clamping and holding operators.

Definition 2.1 (Problem Description) *A problem description is a pair (I, G), where I and G are finite sets of predicates. I is called the initial state and the predicates in I are called features. G is called the set of goals.*

The set of goals G mentions the relations between the objects that must hold. But typically G does not correspond to a complete state. Any state containing G is called a *final state relative to* G (i.e., $G \subset F$ holds).

Example of a Problem Description. Figure 2.2 shows an example of a problem description in the domain of process planning. It consists of 5 goals: to process H, to process the first and second half of U_1 and to process the first and second half of U_2.[2]. Figure 2.2 also shows some of the features in the initial state. Features in the initial state indicate clamping conditions, geometrical relations, cutting tools available, the state of the tool holder and the state of an area (Muñoz-Avila and Weberskirch, 1996c). An example of each is shown in Figure 3.5: the first feature states that the workpiece can be clamped from A_1. The second one that U_1 is a subarea of H. The third one that the cutting tool *lrt* is available. Then, the fourth feature indicates that the tool holder is initially free. Finally, the fifth feature states that initially the area H is unprocessed. Type information is represented as well: in this example, it is stated that the object *lrt* is of type *LeftRTool*.

The solution of a problem is a sequence of actions or plan steps transforming I into a final state relative to G. Plan steps are the basic units of a plan and result from the application of operators:

Definition 2.2 (Applicable Operator) *An operator OP is applicable to a state S if there is a substitution θ such that:*

1. $Prec\theta \subset S$ holds, where $Prec$ are the preconditions of OP.

[2]Because of geometrical properties, to process an undercuts such as U_1, it must be divided in two half parts. Each part is processed separately.

Problem:
Prob3
Goals:
1. processed(H)
2. processedHalf1(U_1)
3. processedHalf2(U_1)
4. processedHalf1(U_2)
5. processedHalf2(U_2)

Features:
1. isClampArea(A_1)
2.*subarea*(U_1,H)
3. available(lrt)
4. toolHolderFree()
5. +unprocessed(H)
...
Objects:
LeftRTool(lrt)
...

Figure 2.2: Part of the symbolic specification of a problem.

2. θ satisfies the constraints of OP.

To check if a constraint is satisfied, the values of the variables must be intantiated according to θ. If, for example, the constraint is *same(x,y)*, then $x\theta = y\theta$ must hold. The operator *HoldTool* is applicable to any state containing the predicate *toolHolderFree()* (see Figure 2.1).

Definition 2.3 (Application of an Operator (Action, Plan Step)) *Let OP be an applicable operator to a state S with a substitution θ. The application of the operator OP to S is the state obtained by the formula $(S - Del\theta\alpha) \bigcup Add\theta\alpha$ if there is a substitution α of the variables in OP not occurring in θ such that:*

1. *$Add\theta\alpha$ and $Del\theta\alpha$ contain only grounded predicates, where Add and Del are the add- and delete-list of the operator.*

2. *$Del^{+}\theta\alpha \subset S$ hold, where Del^{+} is obtained by removing the sign $-$ of each variable predicate in Del.*

3. *$\theta\alpha$ satisfies the constraints of OP.*

Otherwise, the application of the operator OP to S is $\emptyset$.
The application of an operator is called an action or plan step.

In this context, $\emptyset$ indicates an error. For example, if the operator *HoldTool* is applied to an state containing the predicate *toolHolderFree()* and in the current problem no object of type *Tool* is given, the operator is applicable but applying the operator results in $\emptyset$. The reason for the latter is that the constraint *IsOfType(Tool,tool)* cannot be satisfied (see Figure 2.1).

Definition 2.4 (Plan, Solution Plan) *Any sequence, $s_1...s_n$, of plan steps is called a* plan. *That is, s_i corresponds to the application of an operator OP_i.*

Let (I, G) be a problem description and P be a plan, $s_1...s_n$. P is called a solution plan of (I, G), if:

1. *The operator OP_1 is applicable to I, the operator OP_2 is applicable to the state, S_1, obtained after applying OP_1 to I and continuing in this way, OP_n is applicable to the state S_{n-1}.*

2. *$G \subset S_n$ holds, where S_n is the state obtained after applying OP_n to S_{n-1}. That is, S_n is a final state relative to G.*

State-space planners search the solution plans by transforming states; the current state is transformed by selecting one of the applicable operators in pursue of reaching a final state relative to G. The search space that state-space planners traverses is depicted in Figure 2.3. Nodes in this graph are states and arcs are applications of operators. An arc from A to B denotes that there is an operator applicable to the state represented in A such that after applying the operator the resulting state is B. The initial state is one of such nodes (labelled I). Two final states, $F1$ and $F2$, relative to G are shown (i.e., $G \subset F1$ and $G \subset F2$ hold). A solution plan is thus a path from I to a final state relative to G. State-space planners can perform the search in forward direction, from the initial state to a final state, or backwards, from a final state to an initial state, or in both directions at the same time.

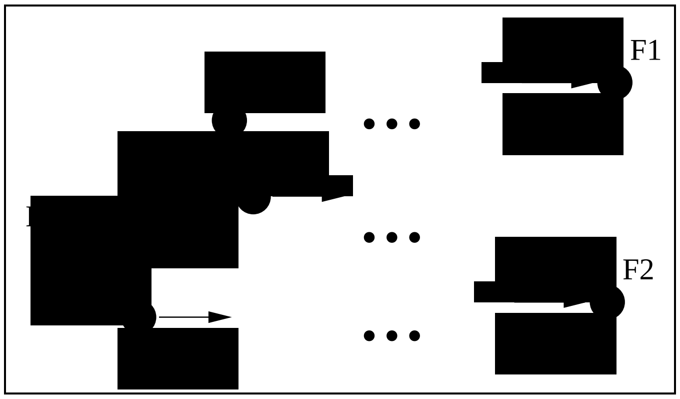


Figure 2.3: Abstract representation of a search space in state-space planning; nodes represent world states and arcs represents actions transforming world states.

2.3 Plan-Space Planning with SNLP

As mentioned in the previous section, state-space planners transform states into states. They naturally follow from the definition of solution plan (Definition 2.4): A sequence of plan-steps such that each step is applicable in the world state obtained after applying the previous step. However, this does not mean that state-space planning is the only way to obtain solution plans. A different approach is followed by so-called plan-space planners. The idea of plan-space planning is to transform partial plans at each step of the planning process. Under this perspective, Figure 2.3

must be reinterpreted: *nodes represent plans and arcs represent actions transforming plans.* The node labelled I represents an initial plan, which is a plan representing the problem and the nodes $F1$ and $F2$ represent solution plans.

One of such plan-space planners is SNLP (for: Systematic Nonlinear Planning). SNLP is a planning algorithm that refines partially ordered plans (McAllester and Rosenblitt, 1991).

Definition 2.5 (Partial-Order Plan) *A partial-order plan is a 4-tuple,* $< S, \rightarrow, \rightarrow_{CL}, B >$*, where:*

- *S is the set of* plan steps. *S always contains two artifical plan steps:* start *and* finish. *Any other plan step in S must be associated with an operator in the domain.*
- *$\rightarrow$ is a set of* links *indicating an order for executing steps in S.*
- *$\rightarrow_{CL}$ is the subset of $\rightarrow$ containing only causal links.*
- *B is a set of constraints on the variables bindings.*

$<_{\rightarrow}$ *denotes the partial order corresponding to the transitive clousure of $\rightarrow$.*

Preconditions and Effects of Steps. Steps in S always contain preconditions and effects. The preconditions (effects) of a plan steps different from *start* and *finish* are the preconditions (effects) of the operator associated with it.

Initial Plan. Once a planning problem (I, G) is given, a so-called initial plan representing the planning problem is constructed. The initial plan contains two artificial plan steps, *start* and *finish*, a single link from *start* to *finish* and no variable bindings. The step *start* contains no preconditions and has as effects the initial features in I whereas the step *finish* contains no effects and has as preconditions the goals in G. Planning proceeds by *establishing open preconditions* and *resolving conflicts.*

Establishment of Open Preconditions. An *open condition* or *subgoal* is a precondition of a plan step such that there is no link from an effect of a step to the precondition. We write $p@s$ to denote that p is a precondition of s. A link can be introduced when the effect of a plan step unifies the open precondition modulo the binding constraints in B. When such a link is introduced, the precondition is said to be established by the plan step. These kinds of links are called *causal links.* Causal links are denoted by $s_1 \rightarrow p@s_2$ indicating that the precondition $p@s_2$ is established with s_1. We distinguish between the case in which s_1 was already in S and the case in which s_1 is a new step (i.e., not in S). In the first case, the establishment is said to be a *simple establishment* whereas, in the second, the precondition is said to be *established with a new step.* When the establishment of the precondition is simple, the constraints making the precondition and the effect unifiable are added to B. When a precondition is established with a new step, the constraints of the associated operator are added to B.

Resolving Conflicts. A conflict or *threat* to a causal link $s_1 \rightarrow p@s_2$ is caused by a third plan step s_3 that has as effect p or $\neg p$ and that is parallel to $s_1 \rightarrow p@s_2$, that is, the following condition does not hold: $s_3 <_{\rightarrow} s1$ or $s_2 <_{\rightarrow} s3$ (see Figure 2.4). If the effect of s_3 is p, in which case we write $s_3 \rightarrow p$, the threat is said to be *positive* and write $s_3 \overset{+}{\longleftrightarrow} (s_1 \rightarrow p@s_2)$. Otherwise, the effect of s_3 must be $\neg p$. In this situation, the threat is said to be *negative* and write $s_3 \overset{-}{\longleftrightarrow} (s_1 \rightarrow p@s_2)$. SNLP solves negative and positive threats. Whereas negative threats are solved to ensure the consistency of the plan, positive threats are solved to ensure the systematicity of the planning process, which can reduce the size of the search space (Kambhampati, 1993). Threats are solved by

- introducing the link $s_3 \longrightarrow s_1$, called a *protection link*, in L (this operation is called a *demotion*),
- introducing the link $s_2 \longrightarrow s_3$, also called a *protection link*, in L (this operation is called a *promotion*), or,
- introducing bindings constraints in B such that the precondition p is not unifiable with the conflicting effect of s_3 (this operation is called a *separation*).

Additionally, no step can be ordered before *start* or after *finish*. Thus, for example, if s_1 is *start* demoting s_3 is not a valid alternative to solve the threat.

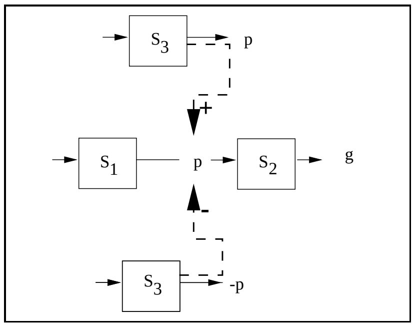


Figure 2.4: Graphical representation of the positive threat $s_3 \overset{+}{\longleftrightarrow} (s_1 \rightarrow p@s_2)$ and the negative threat $s_3 \overset{-}{\longleftrightarrow} (s_1 \rightarrow p@s_2)$.

Solution Plan. At every step of the planning process, partially ordered plans are refined by introducing new links, steps, or ordering constraints. The planning process is finished if the initial plan is refined to a partial-order plan, $Sol = < S, \rightarrow, \rightarrow_{CL}, B >$, containing no open conditions and no threats. This plan representes several solution plans; namely, the set of all totally ordered plans, $P = (S_P, <_P)$, such that:

- $<_P$ is a total order on S extending $<_{\rightarrow}$
- $S = S_P$ holds

Each of these plans is a solution plan for the problem description (I, G) represented in the initial plan (see Definition 2.4). This means that SNLP is correct. We say that *Sol achieves the goals* and refer to *Sol* as a *solution plan* for (I, G). SNLP has been shown to be complete (McAllester and Rosenblitt, 1991), that is, a solution is found if there is one, otherwise, a failure is returned.

Definition 2.6 (Complete Plan) *A partial-order plan is complete if it contains no open preconditions and no threats.*

Example of a complete partial-order plan Figure 2.5 depictes a complete partial-order plan in the domain of process planning. The plan achieves the goal *processed(H)*, processing the horizontal outline.[3] This can be recognized from the plan because *processed(H)* is the only precondition of *finish*. This plan contains three plan steps: to process the horizontal outline (*STEP-3*), to hold the tool *lrt* (*STEP-2*) and to clamp the workpiece from the ascending outline *A1* (*STEP-1*). It also contains four causal links representing the following establishments: $clamp(A_1) \rightarrow$ *processed(H)@finish*, $cs \rightarrow$ *clampFrom(outl-L)@*ms_{ctr}, $hs \rightarrow$ *toolHeld(t-ctr)@* ms_{ctr}, and, $start \rightarrow toolHolderFree@hs$.

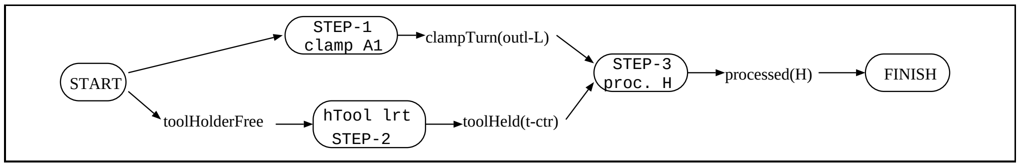


Figure 2.5: Plan for processing (machining) the horizontal outline, *hor*.

2.4 A Planning Theory

There is no planning paradigm that outperforms the others for every domain (Barrett and Weld, 1994; Veloso and Blythe, 1994; Kambhampati et al., 1996a). A planning

[3]In the parlance of mechanical engineering one speaks of *machining* instead of processing.

paradigm may work well for some classes of domains but not for other ones. In (Barrett and Weld, 1994; Kambhampati et al., 1996a) a theory to explain the possible advantages of plan-space planners over state-space planners is presented. *This theory will be used to characterize the circumstances in which the methods presented in this thesis are expected to be adequate.* In this section the main elements of this theory are presented. The first elements introduced in the theory are the classes of plans. We mention two classes:

- **Elastic Protected Plans.** This class of plans is basically formed by the partial-order plans. Plans in which the steps are ordered by precedence relations are said to be *elastic*. Thus, partial-order plans are elastic because they are ordered by $<_{\rightarrow}$. Plans including *interval preservation constraints* (IPC) are said to be *protected*. IPC are 3-tuples of the form $s_1 \overset{p}{-} s_2$ indicating that the condition p must be preserved between s_1 and s_2. Clearly, the IPC $s_1 \overset{p}{-} s_2$ can be seen as the causal link $s_1 \rightarrow p@s_2$. Thus, partial-order plans are elastic and protected.

- **Prefix, Sufix and Prefix, and Sufix Plans.** These are the plans produced by the state-space planners. In a *sufix* plan, the plan steps are contiguous to each other in a chain whose last element is an dummy step representing the final state. In the same way, in a *prefix* plan, the plan steps are contiguous to each other in a chain whose first element is an dummy step representing the initial state.

The second element of the theory is the notion of *trivial serializability*, which will be defined later on. First some definitions are introduced. Consider the plan for machining the center outline shown in Figure 2.5, and, suppose that an additional processing area, for example a drilled hole, is to be machined. If the plan can be extended to machine both processing areas, the plan is said to be serially extensible with respect to the goal corresponding to machining the drilled hole.

Definition 2.7 (Serial Extensibility) *Given a class* $\mathbf{P}$ *of plans, a plan* P *in* $\mathbf{P}$ *achieving a goal* g_1 *is serially extensible with respect to a second goal* g_2 *if there is a refinement of* P *in* $\mathbf{P}$ *achieving both* g_1 *and* g_2.

Notice that backtracking may take place for achieving g_1 and g_2. What the definition of serial extensibility says is that no backtracking should take place in the plan refinement steps introducing steps, links or constraints in P. This is illustrated in the abstract situation depicted in Figure 2.6. The abstract plan was extended by adding new steps and links to a plan achieving g_1 and g_2 (boxes represent plan steps and arcs represent links).

In the same situation as before, if *any* plan for machining the center outline is serially extensible with respect to the goal corresponding to machining the drill, then, the order, first plan for machining the center outline and then for machining the drill, is called a serialization order.

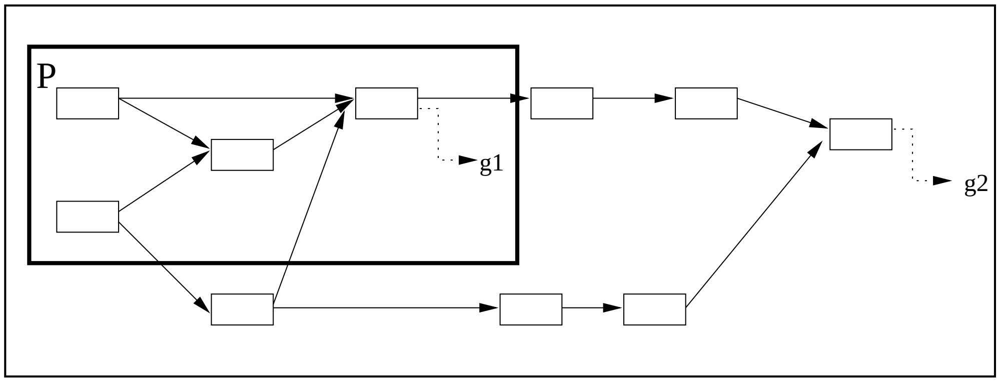


Figure 2.6: Illustration of the notion of serial extensibility.

Definition 2.8 (Serialization Order) *Given a class* $\mathbf{P}$ *of plans and a set of goals,* $g_1, g_2, ..., g_n$*, a permutation* π *on these goals is a serialization order if every plan in* $\mathbf{P}$ *for solving* πg_1 *can be serially extended to* πg_2 *(modulo the class* $\mathbf{P}$*) and any resulting plan can be serially extended to* πg_3 *(modulo the class* $\mathbf{P}$*) and so on.*

Unfortunately, serial extensibility is not a commutative property (Barrett and Weld, 1994; Kambhampati et al., 1996a). Thus, if the order, first plan for machining the center outline and then for machining the drill, is a serialization order, it does not necessarily mean that the opposite order is. If this is the case, the goals are said to be trivial serializable.

Definition 2.9 (Trivial Serializability) *A set of goals,* $g_1, g_2, ..., g_n$*, is trivial serializable, if any permutation of these goals is a serialization order (modulo a class of plans* $\mathbf{P}$*).*

Goals that are trivially serializable can be solved in any order as the subplan achieving any of them can be extended to a subplan achieving two of them and so on. The fact that goals are trivially serializable does not necessarily means that it is easy to obtain a plan achieving the goals. What the definition says is that the effort to obtain a subplan achieving some of the goals will not be lost when extending the subplan to achieve all the goals.

Goals that are trivially serializable in the class of sufix, prefix or sufix and prefix plans are also trivially serializable in the class of elastic protected plans. The essential principle of the theory is that for domains for which goals are trivially serializable in the class of elastic protected plans but not in the class of sufix, prefix or sufix and prefix plans, plan-space planning should be more efficient whereas for domains for which goals are trivially serializable in both classes, state-space planning should be more efficient.

2.5 Static Analysis of a Domain Theory

STATIC is a system that precompiles the domain theory to generate control rules to guide the planning process of a state-space planner (Etzioni, 1993b; Etzioni, 1993a). The precompiled theory is represented as Problem Space Graphs (PSG). A PSG is an AND/OR, bipartite graph representing a part of the search space that may be explored when achieving a goal.

Construction of PSGs. A PSG always contains a single node at the first (top) level. This node contains a predicate g with variable arguments representing several goals (i.e., all possible instantiations of g). Each operator in the domain theory that achieves g is represented as a node at the second level. Each node of the second level is connected with an arc to the node in the first level; they form an OR-tree as any of the operators represented at the nodes can be used to achieve the goal. For each precondition of an operator represented at the second level, a node is added at the third level. Each node n at the third level is connected with a node n' at the second level if n represents a precondition of the operator represented in n. They build an AND-tree as all the preconditions must be achieved for the operator to be applicable. The process is repeated recursively at the subsequent levels. Figure 2.7 represents an abstract PSG. It is assumed that only two operators can achieve g (i.e., *OP-1-1* and *OP-1-2*). Notice that the goal representing the precondition *p1-2* has two parent nodes. This means that *p1-2* is a common precondition of both.

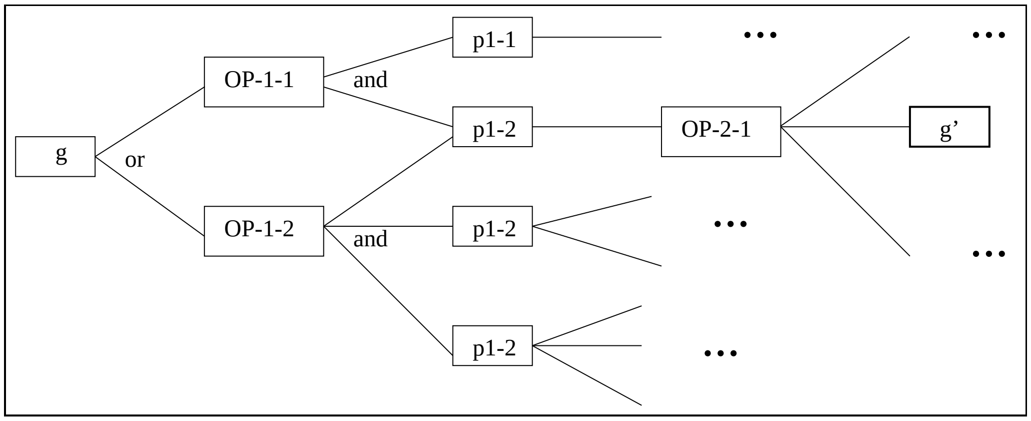


Figure 2.7: Example of an abstract PSG.

Control of Unfolding Process. Clearly, some kind of control must be stated to stop the unfolding process in the PSG. An important condition to stop the unfolding process is when a precondition in a node corresponds to the same predicate represented in an ancestor of the node. For example, no further unfolding will be made below the node containing g' as we are assuming that it has the same predicate as g

(the fact that no further unfolding is possible is represented graphically by the bold square). This condition ensures that the PSG is finite because the branching factor is always finite and the number of predicates is also finite. Another important criterion to stop the unfolding process is based on predefined axioms describing true sentences in the domain. An example of an axiom in the domain of process planning could be: "the tool holder holds a tool or the tool holder is free". This axiom is always true in any situation in this domain. If during the unfolding process, a node is unfolded containing the goal "tool holder is free" and an ancestor of the node contains the goal "hold tool x", no further unfolding is necessary because the axiom guarantees that one of the two goals must be true.

Using PSGs to detect Prerequisite Violations. Etzioni uses the PSG to study the effectiveness of EBL. He also shows that the control rules generated by analyzing the PSG are equivalent to EBL when certain conditions are met. What is important for our work is that Etzioni observed that by using the PSG, rules avoiding *prerequisite violation* can be generated. A prerequisite violation between two goals occurs when achieving one of them first makes it impossible to achieve the other one. There are two possible reasons for a order $g\ h$ to be a prerequisite violation:

- Any subplan achieving g clobbers a precondition necessary to achieve h. That is, there is a precondition p which is necessary to achieve h and any subplan achieving g makes p invalid.
- g needs a precondition which can only be obtained when h is achieved.

Prerequisite Violations as Ordering Constraints. In either situation g must be achieved after h. Notice, that theoretically, the prerequisite violation has no impact in plan-space planners as in these planners the plan steps can be ordered as required. However, the prerequisite violations can be used to determine the ordering constraints $\prec$ of the current problem and, thus, enables the use of dependency-driven retrieval in CAPLAN/CBC in domains where no domain-specific reasoner is available (see Chapter 3). The way $\prec$ can be obtained is straightforward when the prerequisite violations are known:

> If g, h are two goals and the order $g\ h$ is a prerequisite violation, then $h \prec g$ is a valid ordering restriction.

The ordering restriction $h \prec g$ is a valid ordering restriction if for any solution plan h is achieved before g (a formal definition will be given in Chapter 4).

Determining Prerequisite Violations. To determine if there is a prerequisite violation between g and h, the PSGs of g' and h' are generated, where g' and h' are obtained by replacing the arguments of g and h with variables. For each PSG, two sets are generated: the necessary effects and the necessary prerequisites. The necessary effects of a goal are the effects that are always obtained by any subplan achieving the goal. In the same way, the necessary prerequisites are the preconditions that always must be valid to generate any subplan achieving the goal. The rules to determine a prerequisite violation are quite simple: if a necessary effect of g is a necessary precondition of h or a neccesary effect of h clobbers a necessary precondition of g, then $h\ g$ is a prerequisite violation.

Determining the Necessary Effects. The following rules indicate how to generate the necessary effects of a goal by traversing the PSG:

- The necessary effects of a node representing an operator are the effects of the operator united with the necessary effects of its child nodes (i.e., the nodes representing the preconditions of the operator).
- A leaf node has no necessary effects.
- The necessary effects of a node different than a leaf node and representing a precondition of the goal is the intersection of the necessary effects of its child nodes (i.e., the nodes representing the operators that achieve the precondition).

Necessary preconditions can be achieved in a similar way.

2.6 Case-Based Planning

In Prodigy/Analogy the concepts of interacting goals, relevant features and adaptation with replay were introduced (Veloso, 1994), which are now considered an standard in CBP. These concepts will be recalled in this section although the definitions will be reformulated in terms of partially-ordered plans in the sense of SNLP.

The notion of relevance of a feature i in I relative to the particular solution found Sol is motivated by the fact that, whereas a feature may have played a role to find a particular solution plan, it may play no role at all in another solution plan. For example, suppose that an individual called Cesar has to go from his house to the train station. Suppose that Cesar has a car and there is a public bus which passes nearby his house and goes to the train station. The availability of these two ressources can be seen as initial features. If Cesar takes his car, the feature refering to the availability of the bus is completely irrelevant for the plan. Formally, the relevance of a feature can be defined as follows:

Definition 2.10 (Relevant Features) *Given a solution plan $Sol = < S, \rightarrow, \rightarrow_{CL}, B >$ of a problem (I, G), a feature i in I is relevant if there is a causal link of the form start $\rightarrow i@s$ in $\rightarrow_{CL}$ with s in S.*

This means that a feature is considered relevant if it is used to establish the precondition of a plan step in Sol. In the same way that initial features can be discriminated between relevant and irrelevant relative to the particular solution Sol, pairs of goals can be discriminated between interacting and noninteracting. First the notion of connected components will be defined:

Definition 2.11 (Connected Components) *Given a partial-order plan* $Sol = < S, \rightarrow, \rightarrow_{CL}, B >$, *the connected components of* Sol *are the connected components of the graph obtained by viewing each step in* $S - \{start, finish\}$ *as a node and each link in* $\rightarrow$ *as an arc.*

Each connected component represents a complete subplan independent of the others subplans represented in the other connected components. For example, the plan depicted in Figure 2.5 consists of a single connected component because such a decomposition in independent parts is not possible. Given a connected component $Con = < S', \rightarrow >$ of Sol, a complete plan $Sol^{Con} =< S^{Con}, \rightarrow^{Con}, {\rightarrow_{CL}}^{Con}, B^{Con} >$ can be generated, where:

- $S^{Con} = S' \cup \{start^{Con}, finish^{Con}\}$ hold, where $start^{Con}$ contains as effects all the features in I used to achieve preconditions in S' and $finish^{Con}$ contains as preconditions all goals in G achieved by steps in S'.

- $\rightarrow^{Con}$ is the subset of $\rightarrow$ having as source and consumed by steps in S^{Con}.

- ${\rightarrow_{CL}}^{Con}$ is the subset of causal links in $\rightarrow^{Con}$.

- B^{Con} is the subset of B referred by steps in S^{Con}

We can now introduce the notion of interacting goals.

Definition 2.12 (Interacting goals) *Given a solution plan* Sol *of a problem* (I, G), *two goals in* G *interact if they are achieved by the same complete plan represented in a connected component of* Sol.

This means that each connected component of Sol defines a set of interacting goals. Instead of considering $((I, G), Sol)$ as a single case, *each pair of the form* $((I_{Con}, G_{Con}), Sol_{Con})$ *is considered as a case*, where G_{Con} is a set of interacting goals achieved by the complete plan Sol_{Con} represented in a connected component of Sol and I_{Con} is the set of relevant features relative to Sol_{Con}.

Adaptation with Replay. Replay is a method widely used for case adaptation (Veloso and Carbonell, 1993; Bhansali and Harandi, 1994; Blumenthal and Polster, 1994; Ihrig and Kambhampati, 1994; Muñoz-Avila et al., 1994). In this method, instead of storing the solution plan Sol in the case, the derivational trace followed to obtain Sol is stored. The derivational trace is the sequence of planning decisions that were taken to obtain Sol. For example, the derivational path to obtain the plan shown in Figure 2.5 could be: "apply the operator *procc.(H)* to achieve the precondition *processed(H)@finish*", "apply the operator *clamp* A_1 to achieve the precondition *clampTurn(outl-L)@clamp(*A_1*)*" and so on. If the base-level planner is SNLP, the derivational trace contains planning decisions that are proper of plan-space planners such as "order the step *clamp(*A_1*)* after the step *clamp(*A_2*)*". When adapting a case with replay, the derivational trace is followed by applying the planning decisions to the new problem. A planning decision in the derivational is only replayed if no inconsistency will occur as a result of replaying it. For example if the decision says "order the step *clamp(*A_1*)* after the step *clamp(*A_2*)*" and in the current situation *clamp(*A_1*)* $<_{\rightarrow}$ *clamp(*A_2*)* holds, the decision is not replayed because otherwise a cycle in the plan is introduced (i.e., *clamp(*A_1*)* $<_{\rightarrow}$ *clamp(*A_2*)* and *clamp(*A_2*)* $<_{\rightarrow}$ *clamp(*A_1*)* hold simultaneously). In this situation a failure occurs. In Section 5.7 a detailed study of the kind of failures occuring will be made. Adaptation with replay supposes an interaction with the base-level planner. That is, part of the planning process is made by replaying decisions of one or more cases and the other part is made by the base-level planner. A detailed discussion about this issue will be given in Chapter 5.

Part II

The Thesis

Chapter 3

The Problem Solving Cycle in CAPLAN/CBC

In this chapter an overview of the problem solving cycle will be given, the input and output of each phase will be explained and the scope of the knowledge sources will be discussed (a discussion about how our cycle compares to the one of (Aamodt and Plaza, 1994) will be made in Chapter 10). In addition an example of the problem solving cycle is described. A formal and detailed presentation of the different phases of the problem solving cycle will be made in further chapters. These chapters will also discuss the advantages and disadvantages of the methods implemented in each phase, and a characterization of the domains in which using the methods is an adequate choice.

3.1 The Problem Solving Cycle

The problem solving cycle in CAPLAN/CBC consists of four phases: Analysis, Retrieval, Adaptation and Learning. It also considers four sources of knowledge: the domain theory, the case base, domain-specific reasoners and the user. The data flow is depicted in Figure 3.1.

The process starts when a new problem (I, G) is given. I is the set of initial conditions or features and G is the set of goals to be achieved. First, an analysis of the new problem is made. As a result of which a set of ordering constraints $\prec$ to achieve the goals are generated. The ordering constraints together with the problem description form *extended problem descriptions* $(I, G, \prec)$.

Retrieval in CAPLAN/CBC is two fold; it combines static and dynamic retrieval techniques. The static retrieval technique is *dependency-driven retrieval.* Under this approach, a case C always includes its dependencies $<_C$, that is, the order in which the goals are achieved in C. During retrieval, the order $<_C$ is compared against $\prec$ first. Only if certain conditions are met, the initial situations of the case and the problem are compared to determine if C should be retrieved. The dynamic retrieval

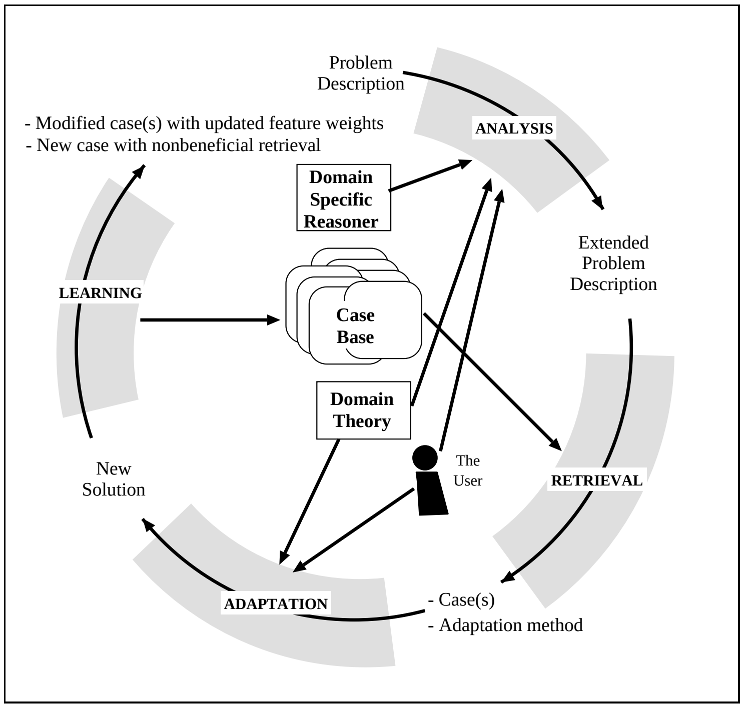


Figure 3.1: Problem solving cycle in CAPlan/CbC.

technique is based on feature weighting. Features in the initial state of the cases have weights. These weights are taken into account when matching the initial situations of the case and the problem. By proceeding in this way one or more cases are retrieved each covering a disjoint subset of goals in the problem. In contrast to the dependencies, the value of the match between the initial states of a problem and a case migh differ depending on the point of time this match is made. The reason for this, is that the feature weights are updated every time a retrieval is made. An additional output of the retrieval phase is the selection of the adaptation method (see Section 7.2).

At the third phase cases are adapted into the new solution by using *complete decision replay.* Base of this strategy is the replay of the structure representing all decisions made when the case was solved and their justifications. In this way, the whole problem solving experience is transfered from the cases into the new situation. After replay, a partial solution is obtained containing the justifications for every decision in the same way as it would have been obtained by the base-level planner. This information is used by the base-level planner CAPlan to prune parts of the solution that are no longer valid because of missing conditions in the new problem and to avoid making decisions that are known from the case to be wrong. In this way the completion effort of the partial solution is reduced in a significant way. The system enables the user to interact during the adaptation process.

After the new solution is obtained from the adaptation phase, the learning phase begins. First, an analysis of the adaptation effort is done to determine if the retrieval of each case is adequate. Next, the set of nonmatched features is selected. If the retrieval is adequate the weights of the features in the set will be decreased. The idea is to reinforce the fact that even when these features were not matched the retrieval was sucessful. If the retrieval failed, the weights of the features in the set are increased. In this situation, the idea is to punish the fact that when these features were not matched the retrieval was a failure. Finally, the new solution is added to the case base if the retrieval is nonbeneficial.

3.2 The Phases of the Problem Solving Cycle

For each phase the input and output will be explained, the knowledge sources, and a brief discussion of how each phase was conceived will be given.

3.2.1 Analysis

Problem descriptions in AI planning have been traditionally defined as a pair (I, G), where I is a set of initial conditions or features and G is a set of goals to be achieved (Fikes and Nilsson, 1971). In complex domains, however, frequently there are intrinsic dependencies between certain key elements. For example, consider the domain of process planning to manufacture mechanical workpieces that are symmetrical with respect to an axis (see Appendix A). In this domain, processing the areas of the workpiece are the goals to be achieved. Due to geometrical restrictions, some areas of the workpiece must be processed before others. A domain-specific reasoner called the geometrical reasoner is used to detect these ordering constraints between the areas to be processed (Muñoz-Avila and Hüllen, 1995; Muñoz-Avila and Weberskirch, 1996b; Muñoz-Avila and Weberskirch, 1996c). Further, these ordering constraints are stated before the planning process begins and they will hold in any manufacturing plan for the workpiece. Thus, the ordering constraints form part of the problem description:

Definition 3.1 (Extended Problem Description) *An extended problem description is a triple $(I, G, \prec)$, where I are the initial conditions or features, G are the goals to be achieved and $\prec$ are the ordering constraints to achieve the goals.*

The input of the analysis phase is a problem description (I, G) and its output is an extended problem description $(I, G, \prec)$. CAPLAN/CBC uses the ordering constraints $\prec$ to improve the accuracy of the retrieval phase. If the information about the ordering constraints $\prec$ is used in an adequate way, the performance of the problem solving process will be improved. The following are the sources that can be used to obtain the ordering constraints $\prec$:

1. A domain-specific reasoner. A typical example of such a reasoner is the geometrical reasoner in the domain of process planning. This domain has been subject

of several studies in AI (Hayes, 1987; Kambhampati et al., 1991; Gil, 1991; Yang and Lu, 1994; Paulokat and Wess, 1994; Nau et al., 1995; Muñoz-Avila and Weberskirch, 1996c). In general, there seems to be a consent about the difficulty of using fully domain-independent techniques to solve problems in this domain (Kambhampati et al., 1991; Nau et al., 1995; Muñoz-Avila et al., 1995; Muñoz-Avila and Weberskirch, 1996c). One reason for this difficulty is the high level of interactions between the plans to process different areas (Hayes, 1987; Paulokat and Wess, 1994). The geometrical reasoner precomputes some of these interactions that should reduce the search effort of the generic problem solver.

2. The user. CAPLAN/CBC considers the user as a possible source for the ordering restrictions between the goals. The user can state some ordering constraints because of his/hers better overview of the problem. There is, however, a restriction that must be taken into account: The generic problem solver always must obtain plans meeting the ordering restrictions even if they are not given explicitly (see Chapter 4). That is, the ordering restrictions if used in an adequate way, enables the generic problem solver to find a solution more rapidly. But if they are not given, the generic problem solver should find a solution meeting the ordering restrictions.

3. A domain-independent analyst. Etzioni observed that by using the domain theory some information about the solution can be precompiled (Etzioni, 1993b; Etzioni, 1993a). Part of the precompiled information are ordering constraints to achieve the goals. For example, in the Schedworld domain, it can be precompiled that shaping a surface into cylindrical form should be performed before polishing it (Etzioni, 1990). Base of this method is the construction of a tree-like structure on which a static analysis of the interactions can be made (see Section 2.5). The relevance for CAPLAN/CBC of this work is the fact that in several domains the ordering constraints $\prec$ can be computed automatically. Thus, *the whole problem solving cycle in* CAPLAN/CBC *is domain-independent in these domains.*

3.2.2 Retrieval

The input of the retrieval phase is an extended problem description $(I, G, \prec)$. One particularity of CAPLAN/CBC is that the dependencies between the goals are represented explicitly in the cases. Informally, the notion of dependencies can be defined as follows (the formal definition will be given in Chapter 4):

Definition 3.2 (Goal Dependencies) *Let g, h be two goals achieved in a case C. The goal h is said to depend on the goal g and write $g <_C h$ if there are two steps s_g, s_h in the solution of C achieving g and h such that s_h occurs after s_g in Sol.*

When a new problem $(I, G, \prec)$ is given, all cases in the case base are considered as candidate cases. The retrieval strategy in CAPLAN/CBC is dependency-driven in that candidate cases are eliminated because of violations to a condition regarding their dependencies $<_C$. The condition can be stated as follows:

Definition 3.3 (Order Inclusion Condition) *A case C meets the ordering inclusion condition with respect to a problem $(I, G, \prec)$ if there is a substitution θ such that*

1. *$G_C\theta \subset G$ (where G_C are the goals achieved in C).*
2. *For every pair of goals $g_1, g_2 \in G_C$, if $g_1\theta \prec g_2\theta$ holds, then $g_1 <_C g_2$ must also hold.*

The rationale behind the first requirement of the order inclusion condition is to avoid the risk that additional goals achieved by the case interact negatively with the goals of the problem. Condition 2 reflects the dependency-driven retrieval strategy: the dependencies of the case must *extend* the ordering constraints of the problem (informally written: $\prec \subseteq <_C$). In Chapter 4 a weakened form of this condition will be stated, which is actually the form used in CAPLAN/CBC.

Once cases not meeting the order inclusion condition are eliminated from the list of candidate cases, CAPLAN/CBC proceeds to compare the initial state of the candidate cases against I. As in PRODIGY/ANALOGY only relevant features are taken into account by using the foot-printing process to discern between relevant and nonrelevant features of a case (see Section 2.6).

A distinguished characteristic of CAPLAN/CBC is that features are not only classified as relevant and not relevant, but they are ranked by associating weights to them. The weight, $\omega_{i,C}$, of a feature i depends on the particular case C in which the feature occurs. A feature may be relevant for two cases, but whereas it may have a high rank in one of them, it may have a low rank in the other one. That is, the feature might be very important for one case but not so important for the other one. The similarity metric in CAPLAN/CBC takes into account the feature weights:

Definition 3.4 (Weighted Similarity Metric) *The weighted similarity metric between a case C and a problem P, $sim^{wg}(C, P)$, is defined as:*

$$sim^{wg}(C,P) = \begin{cases} \sum_{i \in I \bigcap_\theta I_C} \omega_{i,C} & : \ G_C\theta \subset G \\ 0 & : \ otherwise \end{cases}$$

where $I \bigcap_\theta I_C$ denotes the set of all features in I_C matching a feature in I with a substitution θ.

Testing all candidate cases to find the one that is most similar has a prohibitive cost. For this reason, CAPLAN/CBC searches for the first candidate case that meets the following condition:

Definition 3.5 (Weighted Retrieval Condition) *Given a problem P, a case C meets the weighted retrieval condition, $SIM^{wg}(C,P)$, if and only if $G_C\theta \subset G$ and*

$$(sim^{wg}(C,P)/sim^{wg}(C,C)) \geq thr_{ret}$$

where thr_{ret} is a predefined threshold, called the retrieval threshold.

In other words, the weighted retrieval condition is met if the weighted proportion of features in I_C that match features in I modulo θ is greater than the retrieval threshold thr_{ret}.

In summary, a case is retrieved if it meets the order inclusion and the weighted retrieval conditions. In addition, the substitution θ in both conditions must be the same. In Chapters 4 and 7 an indexing structure of the case base will be presented that allows CAPLAN/CBC to test both conditions in a more efficient way than testing these conditions sequentially on all the cases. To retrieve multiple cases CAPLAN/CBC follows the same top down strategy as in PRODIGY/ANALOGY by pursuing to cover the set of goals with as few cases as possible.

3.2.3 Adaptation

The input of the adaptation phase is the list of retrieved cases and the output is a solution plan for the problem $(I, G, \prec)$.

The adaptation method conceived and implemented in CAPLAN/CBC is called complete decision replay. This method is fundamented on the way the base-level planner CAPLAN is built (Weberskirch, 1995).[1] To represent knowledge about plans and contingencies that occur during planning, CAPLAN is built on the generic REDUX architecture (Petrie, 1991a; Petrie, 1991b). The REDUX architecture represents relations between goals and operators and between operators and subgoals. In the parlance of REDUX a *decision* is made when an operator is applied to achieve a goal. Decisions are represented as a subtree in the *subgoal graph*. Subgoal graphs represent basic dependencies between goals and subgoals as well as between subgoals and decisions. Figure 3.2 depictes a decision.

REDUX provides a TMS algorithm (Doyle, 1979) to maintain the dependencies. The nodes in the TMS represent different aspects of the truth maintenance process such as the validity of a goal. Changes in a node are propagated through the TMS. CAPLAN maps the concepts of partially ordered plans into the REDUX structure. As for now, this map can be imagined as a straightforward map from planning goals to REDUX goals and from planning operators to REDUX operators (a more detailed description will be given in Chapter 5). A key aspect in CAPLAN is that the *justifications* of all decisions (valid and rejected) are always mantained. A justification has the form $\{a_1, a_2, ..., a_n\}$, where a_i is a constraint. For example, consider a

[1]The development of CAPLAN does not form part of this thesis. The overview presented here and the more detailed explanation in Chapter 5 are made as basis to explain the complete decision replay adaptation method.

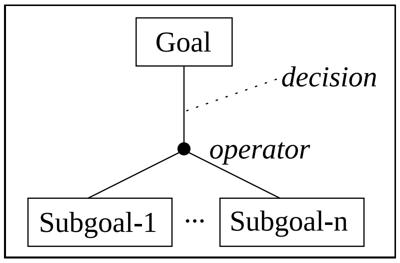


Figure 3.2: Tree representation of a decision in REDUX.

decision corresponding to the application of an operator that requires $x = y$ to be true, where x and y are variables. If the decision is valid, an example of a possible justification is $\{x = 1, y = 1\}$. An example of a justification if the decision is invalid is $\{x = 1, y = 0\}$.

CAPLAN/CBC store as part of a case, the goal graph and the justifications of every decision made when the case was solved (Muñoz-Avila and Weberskirch, 1996b; Muñoz-Avila and Weberskirch, 1996a). Complete decision replay consists of three steps:

1. Replay of the goal graph. Valid decisions are reconstructed relative to the new problem. The reconstruction of the valid decisions correspond to replay in the usual way (Veloso, 1994): The operator applied to achieve a goal in the case is applied to achieve the corresponding goal in the new problem.

2. Replay of the justifications. CAPLAN/CBC pursues to reconstruct the justifications of every rejected decision relative to the new situation. During this process, two situations may occur:

    1. All the constraints in the justification can be reconstructed. This means that the decision can be rejected in the new situation.

    2. One or more constraints cannot be reconstructed. In this situation, it cannot be guaranteed that the decision can be rejected in the new situation. Thus, the decision will need to be explored by the base-level planner CAPLAN at the third step of the adaptation process.

3. Completion of the partial solution. The partial solution plan obtained after the performing the previous two steps is completed by the base-level planner CAPLAN. That is, first-principles planning is performed.

During the completion phase, CAPLAN will not make decisions that were rejected in the case provided that their justifications were successfully reconstructed. As a result, the search space will be reduced. The gains in efficiency are due to the fact that CAPLAN needs to make a consistency check (corresponding to a constraint propagation process) every time a decision is made. In contrast, the reconstruction process of the justifications (step 2), no consistency check is necessary.

A second aspect to consider is that finding the parts in the plan that are no longer valid because of missing conditions can be made efficiently. This is possible due to the dependencies represented in the subgoal graph.

Finally, as the subgoal graph has been reconstructed, facilities inherent to CAPlan such as interactivity and dependency-directed backtracking (Weberskirch, 1995) can be performed during the step 3 of the adaptation process. In particular, the user may remove parts of the partial solution plan obtained after step 2 and/or indicate which operators to select to achieve a goal.

The merging method for adapting several cases used in this work is based on the one proposed in (Veloso, 1994) and implemented in the context of plan-space planning as reported in (Ihrig and Kambhampati, 1996a). The motivation of this method is to avoid redundancy during multi-case replay; before adding a new step to achieve a goal, the system checks if no existing step can achieve the goal. The new step is added only if no such a step exists. This method has been implemented in the context of complete decision replay as will be explained in Chapter 5. A trade-off related to the efficiency gains that we found is associated with this method is studied in Chapter 8.

Note about the word "dependency". The word dependency has two meanings in this thesis. First, in the context of the dependency-driven retrieval technique, the word *dependency* denotes the partial order in which the goals are achieved in a particular solution plan (see Definition 3.2). Second, in the context of complete decision replay, it denotes the relations between the planning objects represented in the subgoal graph.

3.2.4 Learning

Once a solution has been found, CAPlan/CbC performs an analysis to determine if the solution is to be stored as a new case. In addition the feature weigths of the retrieved case are updated.

A definition of failure has been stated in the literature (Ihrig and Kambhampati, 1996a). This definition can be stated as follows:

Definition 3.6 (Retrieval Failure, Adequate Retrieval) *Given a solution plan Sol of a problem P obtained by adapting a case C, the retrieval of C is a failure with respect to P and Sol if at least one decision replayed from C was revised by the first-principles planner to obtain Sol. Otherwise the retrieval of C is said to be adequate.*

This definition says that the retrieval of a case is considered adequate if the partial solution obtained after replay can be extended to a complete solution without having to revise any of the decisions that were replayed from the case. This definition suggests an strategy that is followed by the base-level planner CAPLAN to complete the partial solution: it will try to extend the partial solution first. Decisions made to obtain the partial solution are only revised if no extension is possible. However, there are two circumstances that need to be considered:

1. CAPLAN/CBC performs complete decision replay instead of standard replay. Thus, even if the retrieval is a failure, the effort to complete the solution might not be large because of the reconstruction of the failed attempts reduces the search space (see Section 3.2.3).

2. The retrieval of the case is adequate but the completion effort is large. During the completion process, the first-principles planner may not require to revise a replayed decision but still the completion effort can be large.

These circumstances are related to the *benefit* of retrieving the case. The fact that the retrieval of the case is adequate does not necessarily implies that retrieving it results in a benefit to the performance of the problem solving process. A difficulty of considering the benefit is that there is no domain-independent procedure to determine it. To exactly measure the benefit of solving the problem with the case-based planner it would be necessary to know the effort required by the first-principles planner to solve the problem. This is, of course, not a feasible possibility. Instead, CAPLAN/CBC introduces a heuristic measure to determine the benefit of the retrieval:

Definition 3.7 (Beneficial Retrieval) *Given a solution plan Sol of a problem P obtained by adapting a case C, then the retrieval of C is beneficial with respect to P and C if:*

$$searchSpace(Sol)/searchSpace(PSol) \leq thr_{ben}$$

where PSol *indicates the partial solution obtained after replay,* thr_{ben} *is a predefined threshold and* searchSpace(Sol) *returns the size of the search space explored to obtain the plan Sol. The threshold* thr_{ben} *is called the benefit threshold.*

The function *searchSpace(Pl)* counts the number of decisions made to compute the plan *Pl* (see Chapter 6). Thus, *searchSpace(Sol)* $\geq$ searchSpace(PSol) always holds. The benefit threshold thr_{ben} determines how eagerly cases will be learned. If, for example, $thr_{ben} = 1$, then anytime a decision is made to find *Sol*, the retrieval is considered nonbeneficial. If the value of thr_{ben} is set to 2, the retrieval is nonbeneficial if the size of the search space explored to complete the case is at least as big as the number of replayed decisions.

There are situations in which the retrieval fails but it is beneficial or the retrieval is adequate but nonbeneficial. The first situation means that some replayed decisions were revised to obtain a solution but the effort to complete the solution was in acceptable limits. The second one means that no replayed decisions were revised but the effort to complete the solution was too large. These situations are considered in CAPLAN/CBC's policy to add new cases:

$\boxed{\text{CAPLAN/CBC adds a solution as a new case if and only if the retrieval of the case is nonbeneficial with respect to the problem and the solution.}}$

If CAPLAN/CBC determines that the new solution is to be stored as a case, the dependencies between the goals are computed. They constitute the main discrimination criteria of the new case (see Chapter 4).

The information about the adequateness or failure of the retrieval of a case C is used in CAPLAN/CBC to update the feature weights. First, the set, *NMatch*, of features of C that do not match features of the new problem is computed. In addition each case has two variables: k^C and f^C indicating the number of adequate and failed retrievals respectively. These two variables are the most important factors to determine the *incremental factor*, $\triangle_{k^C,f^C}$, that is used to update the feature weights.[2] The principle to compute $\triangle_{k^C,f^C}$ is based on the relation between k^C and f^C: the larger the proportion of k^C to f^C, the smaller $\triangle_{k^C,f^C}$. As a result, in cases where k^C is much larger than f^C, the changes in the feature weights are smaller. The rationale behind this principle is to maintain the distribution between feature weights for the cases in which statistically most of the times the retrieval is successful and to change this distribution in a significant way for cases in which not (see Chapter 6). Figure 3.3 shows the algorithm to update the feature weights. The value $\omega_{i,C}$ denotes the weight of the feature i in C and the boolean function *failedRet(C,Prob,Sol)* returns true if and only if the retrieval of C fails with respect to the problem *Prob* and the solution *Sol*.

```
updateFeatureWeights($C, Prob, Sol$)
1. If failedRet(C,Sol,PSol)
     For-each $i \in NMatch$
       $\omega_{i,C} = \omega_{i,C} + \triangle_{k^C,f^C}$
2. Else
     For-each $i \in Expl$
       $\omega_{i,C} = \omega_{i,C} - \triangle_{k^C,f^C}$
```

Figure 3.3: Algorithm to update the feature weights.

If several cases are retrieved the evaluation of adequateness or failure is done case by case (see chapter 6). The benefit of the retrieval is evaluated by taking into account the contributions of all retrieved cases (see Chapter 8).

In summary, the input of the learning phase consists of the extended problem description, the solution plan obtained from the adaptation phase, the retrieved

[2]Other factor considered is the number of initial features in the case (see Chapter 6).

cases and the current case base. The output is the updated case base. There are two possible updates for the case base: the new solution is added (it occurs if the retrieval is nonbeneficial) and the feature weights of the retrieved case are updated (it occurs always).

3.3 A Case in CAPLAN/CBC

The information stored in a case is used by CAPLAN/CBC either during retrieval or during the adaptation process. For this reason the information contained in the cases can be classified as follows:

Adaptation Information. This information is contained in the subgoal graph, which is the REDUX-based structure that represents a partially ordered plan in the base-level planner CAPLAN. The subgoal graph describes all decisions (i.e., valid and invalid) made as the problem was solved and their justifications.

Retrieval Information. The goals achieved in the case and their dependencies $<_C$, i.e. the order in which the goals were achieved in the plan, form part of every case. The features, their weights and the number of adequate and failed retrievals also form part of the case.

3.4 The Knowledge Sources of the Problem Solving Cycle

Two of the knowledge sources are necessary for CAPLAN/CBC: the domain theory and the case base. The other two, the domain-specific reasoners and the user, are motivated by the fact that in many real-world applications these information sources are available and must be taken into account.

3.4.1 The Domain Theory

CAPLAN/CBC as well as other case-based and first-principles planners assume that a symbolic specification modelling the world is available. The symbolic specification models actions that transform the world. Actions are modelled with *operators*. *Operators* have preconditions, indicating the conditions that must hold to apply them. The effects of an operator indicate the changes to the world after the operator has been applied. In addition, the only changes that occur to the world are the ones stated in the effects of the operator (Fikes and Nilsson, 1971). Section 2.1 formally defines these concepts.

The level of granularity of the symbolic specification depends on the particular domain and the purpose of the system. In the domain of process planning, for example, a specification has been developed that includes 27 operators (see Appendix

A). The symbolic specifications of other domains consists of fewer operators (for example the blocks world consists of 4).

As mentioned in section 3.2.1, a static analysis can be also performed to the domain theory to predetermine ordering constraints to achieve the goals. In CAPlan/CbC this is conceived to be performed in the analysis phase. Section 2.5 resumes this procedure.

3.4.2 The Case Base

Whereas the domain theory is a common knowledge source for both the first-principles planner CAPlan and the case-based planner CAPlan/CbC, the case base is a knowledge source exclusive of the second one. Typically, the case base not only consists of a collection of cases but of an indexing structure that enables the case-based planner to evaluate the similarity metric more rapidly. In CAPlan/CbC the indexing structure consists of three levels: at the top level cases are discriminated by the dependencies between the goals. At the second level cases are discriminated by their weighted features. The third level is the collection of cases.

In domain-specific, case-based planners such as CHEF (Hammond, 1986) the case base contains part of the domain theory. For domain-independent, case-based planners such as CAPlan/CbC the case base contains meta knowledge about how the symbolic specification modelling the world has been used to solve problems.

3.4.3 Domain-Specific Reasoners

There are domains that due to their complexity, it is not feasible to assume that any generic problem solver is capable of solving all problems in reasonable time. For example, in the domain of process planning, the difficulty of solving problems solely with a generic planner has been observed (Kambhampati et al., 1991; Nau et al., 1995; Muñoz-Avila et al., 1995; Muñoz-Avila and Weberskirch, 1996c). Instead, for this domain pure domain-specific techniques (Hayes, 1987) or mixed strategies involving the integration of domain-specific and domain-independent techniques have been suggested (Kambhampati et al., 1991). The latter is the strategy followed in CAPlan/CbC (Muñoz-Avila and Weberskirch, 1996b); a domain-specific reasoner is used to compute ordering constraints between the goals. These ordering constraints are computed based on geometrical interactions, which are known to be a major difficulty in this domain (Hayes, 1987; Paulokat and Wess, 1994). CAPlan/CbC considers the integration of domain-specific reasoners as part of its analysis phase; they precompute ordering constraints between the goals.

3.4.4 The User

The user is an important source of information, which is ignored frequently (particularly in CBP). Given his or her better understanding of the problem, the user may state ordering constraints between the goals. That is, the user may contribute to the analysis phase. Another important aspect is the interactivity of the planning process. One of the basic premises for the construction of the first-principles planner CAPLAN is to support user interactions (Weberskirch, 1995; Weberskirch and Paulokat, 1995). With the complete decision replay method, CAPLAN/CBC reconstructs the goal graph, which is the base for CAPLAN to support user interaction. This means that CAPLAN/CBC supports an interactive adaptation of the case as the user may prune out parts of the replayed case or re-state the validity of initial conditions after replay has taken place.

3.5 Example of a Problem Solving Cycle

The example occurs in the domain of process planning to manufacture mechanical workpieces. A five goal problem in this domain will be given and a case achieving three of the goals and partially matching the features of the problem will be retrieved. Complete decision replay will be used to adapt the case and finally, CAPLAN/CBC will learn from this problem solving episode.

3.5.1 The Domain of Process Planning

Figure 3.4 shows an example of a workpiece ("a long shaft"). A planning problem in this domain is given by a geometrical description of a workpiece and of the stock (raw material). The description of a workpiece is built up from geometrical primitives like cylinders, cones and toroids that describe monotone areas of the outline, possibly augmented by features (threads, undercuts, surface conditions). For such a planning problem a sequence of processing operations is to be found that will machine the workpiece considering available resources (i.e. tools, machines) and technological constraints related to the use of these resources. The process begins with clamping a piece of raw material on a lathe machine that rotates it at a very high speed. In most situations, the outline of the workpiece cannot be machined in one step but repeated cutting operations are necessary to cut the difference between the raw material and the workpiece in thin horizontal or vertical layers.

The processing of each area conforming the workpiece is a goal. In Figure 3.4 seven of these processing areas are shown: the two ascending outlines *A1* and *A2*, the horizontal outline *H*, the two sides *S1* and *S2*, and the two undercuts U_1 and U_2. As mentioned before, the ordering restrictions in this domain are stated by a geometrical reasoner. In particular, the geometrical reasoner determines that the processing area *H* must be manufactured before the processing areas U_1 and U_2. These constraints must be met by any partially ordered plan for manufacturing that workpiece.

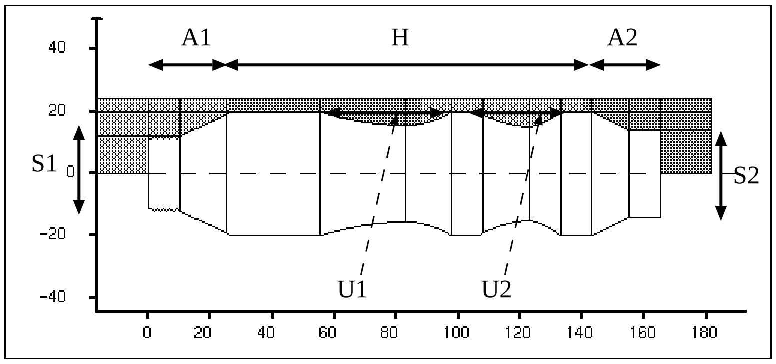


Figure 3.4: Display of a rotational symmetrical workpiece.

3.5.2 The New Problem

Suppose that a new problem is given that consists of processing the areas H, U_1 and U_2 of the workpiece shown in Figure 3.4. This problem is shown in Figure 3.5 (this problem description was discussed in detail in Section 2.2).

Problem:
Prob3
Goals:
1. processed(H)
2. processedHalf1(U_1)
3. processedHalf2(U_1)
4. processedHalf1(U_2)
5. processedHalf2(U_2)

Ordering Constraints:
$1 \prec 2$
$1 \prec 3$
$1 \prec 4$
$1 \prec 5$

Features:
1. isClampArea(A1)
2. subarea(U_1,H)
3. available(lrt)
4. toolHolderFree()
5. +unprocessed(H)

...

Objects:
LeftRTool(lrt)
...

Figure 3.5: Part of the symbolic specification of a problem.

3.5.3 The Case

Suppose that a case, $C5$, is available in which the area H and the two half parts of U_1 of the same workpiece as in the problem *Prob3* are processed. The part of the case which is taken into account during retrieval is shown in Figure 3.6. The goals and their dependencies are shown (the solution plan will be shown later). In addition the statistical information about the number of adequate and failed retrievals, k_{C5}

Case:
C5
Goals:
1. processed(H)
2. processedHalf1(U_1)
3. processedHalf2(U_1)

Dependencies:
$1 <_C 2$
$1 <_C 3$
$2 <_C 3$
Statistics:
k_C: 0
f_C: 0

Features:
1. isClampArea(A1), $\omega_1 = 1$
2. subarea(U_1,H), $\omega_2 = 1$
3. available(lrt), $\omega_3 = 1$
4. available(rrt), $\omega_4 = 1$
5. toolHolderFree(), $\omega_5 = 1$
6. +unprocessed(H), $\omega_6 = 1$

...
Objects:
LeftRTool(lrt)
RightRTool(rrt)
...

Figure 3.6: Part of the case that is considered during retrieval.

and f_{C5} respectively, is maintained. In this situation, we assume that the case has not been retrieved before. Notice that all the feature weights are 1. In addition, we assume that in contrast to problem *Prob3*, an additional cutting tool *rrt* is available (feature 4).

As discussed in Section 3.3, subgoal graphs and not plans are stored in the cases. However, in this example we will suppose that *C5* stores the plan depicted in Figure 3.7 for the sake of simplicity (see Chapter 5 for a concrete example of a subgoal graph). Steps 3, 4 and 8 process H and the two half parts of U_1 respectively. Thus the goal 1 is achieved before the goals 2 and 3 as expressed by the dependencies (Figure 3.6). The plan begins by clamping the workpiece from the area $A1$ and holding the cutting tool *lrt* (steps 1 and 2). After processing H (step 3), the first half of U_1 is processed (step 4). Finally, the cutting tool *rrt* is held (step 6) before processing the second half of U_1 (step 8). The figure shows also the three alternatives to process the second half of the area U_2. The first alternative is to use a certain type of tool (i.e., a "right cutting tool"). This alternative was selected in that plan (i.e., *step-6*). The second alternative is to mount the workpiece from the ascending area $A2$ (i.e., *step-6'*). we assume that this alternative has not been pursued during the planning process. The third alternative is to mount the workpiece from the side $S2$ (i.e., *step-6"*). This alternative requires that the corresponding side contains a hole of certain dimensions. We also assume that this alternative has been pursued during the planning process and that it has been rejected because the workpiece shown in Figure 3.4 does not contain any hole in $S2$.

3.5.4 Retrieval

Given a new problem, CAPLAN/CBC retrieves a case if the order inclusion condition and the weighted retrieval condition are met (see Definitions 3.3 and 3.5). Notice

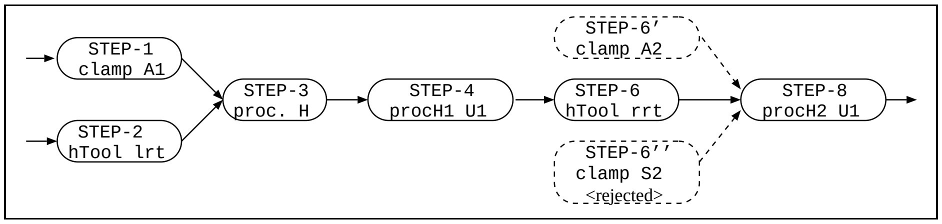


Figure 3.7: Solution plan of the case.

that the goals of the case $C5$ match a subset of goals of the problem *prob3* with the identity substitution $\{H \rightarrow H, U_1 \rightarrow U_1\}$. As the ordering constraints are extended by the dependencies of the case, the inclusion condition is met. Thus, the case meets the static requirement. Finally the weighted retrieval condition is also met because the only unmatched feature in the case is feature 4 (i.e., *available(rrt)*) and the retrieval threshold for the domain of process planning is set to 75%. Thus, the case meets also the dynamic requirement. As required, the two conditions are met with the same substitution (i.e., the identity substitution). Thus, the case $C5$ is retrieved.

3.5.5 Adaptation

Complete decision replay reconstructs the subgoal graph as well as the justifications for every decision. The resulting plan is depicted in Figure 3.8. Most of the decisions were successfully replayed. Step 6 was replayed as well but the precondition regarding the use of a tool of type "right cutting tool" is left to the first-principles planner (this situation is depicted by a question mark, "?", adjacent to step 6). In addition, the fact that *step-6"* is invalid in the new situation has been also reconstructed. Intuitively, given that the workpiece in the case and of the problem are the same, the justification for rejecting that step, i.e. there is no hole on $S2$, can be reconstructed in the new situation.

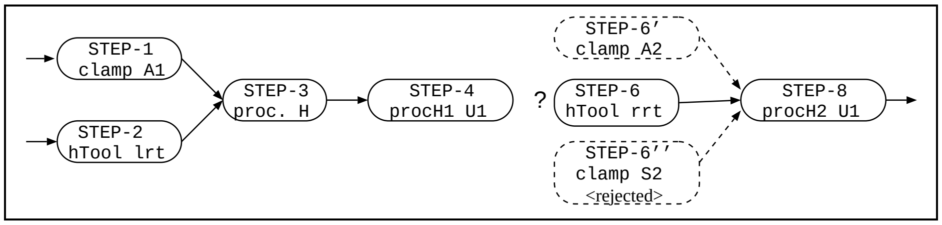


Figure 3.8: Partial solution obtained after complete replay.

The partial solution is then completed by the first-principles planner. The solution plan of the new problem is depicted in Figure 3.9. The step 6 has been rejected because there is no possibility of generating the cutting tool of the required type (this can only be stated as a feature in the initial condition). Given that *step-6"* is also known to be invalid the only remaining alternative is to apply step *6'*. The goals 4 and 5 are achieved by the steps 5 and 10. Because of the plan-space planning paradigm implemented in the base-level planner CAPLAN, these steps can be interleaved in the plan without having to revise any replayed step.

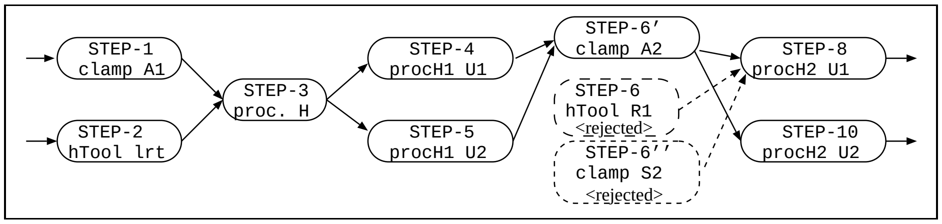


Figure 3.9: Solution obtained after the adaptation process.

3.5.6 Learning

The last phase in the problem solving cycle of CAPLAN/CBC is the learning phase. At this phase CAPLAN/CBC evaluates if the retrieval was successful and beneficial (see Definitions 3.6 and 3.7).

The retrieval failed because at least one decision replayed from the case was revised during the completion process (i.e., step 6). Following the algorithm *update-FeatureWeights* shown in Figure 3.3, the weight of the feature 4, *available(rrt)*, is incremented by the factor $\triangle_{0,0}$.

Finally, the benefits of the retrieval are evaluated. The benefit threshold thr_{ben} is set to 2. That is, the retrieval is nonbeneficial if the size of the search space explored during the completion process is at least as large as the size of the partial solution obtained after replay. In this particular situation the retrieval is beneficial as the completion effort is small. Thus, the solution of the new problem is not added as a new case in the case base. Notice that this is an example of a failed and beneficial retrieval. This situation is not unusual as we will see in the experiments situation (see Chapter 9).

Chapter 4

Dependency-Driven Retrieval

In many domains, more information about a problem is known than just the initial state and the goals to be achieved. More concrete, in these domains ordering restrictions to achieve the goals are also known. Ordering restrictions can be obtained from different sources such as domain-specific reasoners, domain-independent analyst, or the user (see Section 3.2.1 for a general discussion about these sources and Section 2.5 to see how a domain-independent analysts can be used to generate the ordering restrictions).

When these ordering restrictions are available, a natural question to ask is how can they be used to improve the performance of the generic planner. In the context of CBP, CAPLAN/CBC answers this question by taking them into account during the retrieval phase with a technique called dependency-driven retrieval. This technique is developed in CAPLAN/CBC to improve the accuracy and the performance of the retrieval phase (Muñoz-Avila and Hüllen, 1995; Muñoz-Avila and Weberskirch, 1996b). Three issues are studied in this chapter:

- Definition of a retrieval assessment that takes into account the ordering restrictions.
- Design of an indexing structure for the case base based on the retrieval assessment.
- Implementation of a retrieval procedure based on the indexing structure to evaluate the retrieval assessment efficiently.

In addition, this chapter discusses the characteristics that the domain should have for dependency-driven retrieval to be useful. Dependency-driven retrieval constitutes the static retrieval technique supported in CAPLAN/CBC. In further chapters, a dynamic retrieval technique is studied and the integration of these two techniques is explained.

4.1 Extended Problem Descriptions

In CAPlan/CbC the ordering restrictions are considered part of the so-called extended problem descriptions. Before formally defining them some notation must be introduced:

Definition 4.1 (establisher(g,P)) *Given a complete plan $P = (S, \rightarrow, \rightarrow_{CL}, B)$ and a goal g achieved in P, establisher(g, P) denotes the step $s_g \in S$ such that:*

1. *s_g achieves g, and*
2. *There is no plan step $s \in S$ such that $s_g <_{\rightarrow} s$ and s clobbers g.*

In other words, *establisher(g,P)* is the maximal, nonclobbered step in P achieving g.

Proposition 4.1 establisher(g,P) *is unique.*

Proof. The good definition of *establisher(g,P)* follows from the completeness of P (i.e., P contains no open preconditions and no unsolved threats); there is at least one plan step meeting conditions 1 and 2. Otherwise, either the precondition of *finish* representing g remains open if condition 1 does not hold, or, a negative threat remains unsolved if condition 2 does not hold: if there is a step $s \in S$ with $s_g <_{\rightarrow} s$ and s clobbers g, then the threat $s \overset{-}{\longleftrightarrow} (s_g \rightarrow g@finish)$ occurs. Further, there cannot be two maximal steps achieving g meeting conditions 1 and 2 because otherwise two positive threats occur: if s_g and s'_g are two steps meeting conditions 1 and 2, then the threats $s_g \overset{+}{\longleftrightarrow} (s'_g \rightarrow g@finish)$ and $s'_g \overset{+}{\longleftrightarrow} (s_g \rightarrow g@finish)$ occur. This situation is illustrated in Figure 4.1.■

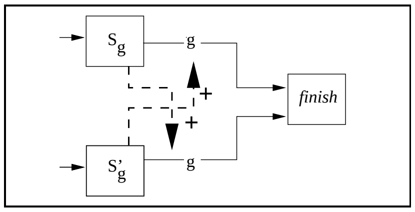


Figure 4.1: Graphical representation of two maximal steps achieving g.

Definition 4.2 ($PLAN(I,G)$,$PLAN_{\prec}(I,G)$) *Given a problem description, the set of all partial-order plans solving (I, G) is denoted by $PLAN(I, G)$.*

If $\prec$ is a partial order on G, then $PLAN_{\prec}(I, G)$ denotes all plans $(S, \rightarrow, \rightarrow_{CL}, B)$ in $PLAN(I, G)$ such that for all g_1 and g_2 with $g_1 \prec g_2$, establisher$(g_1, Pl) <_{\rightarrow}$ establisher(g_2, Pl) holds.

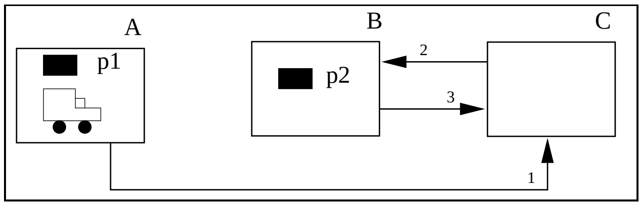


Figure 4.2: A situation in the logistics transportation domain.

Informally, $PLAN_{\prec}(I,G)$ denotes all plans $(S, \rightarrow, B)$ solving (I,G) such that $\prec \subseteq <_{\rightarrow}$ holds. That is, for which $<_{\rightarrow}$ extends $\prec$.

Proposition 4.2 *$PLAN_{\prec}(I,G) \subset PLAN(I,G)$ holds.*

This proposition follows from the fact that any plan in $PLAN_{\prec}(I,G)$ is a solution for (I,G). The opposite, however, does not hold. Consider, for example, the initial situation illustrated in Figure 4.2 in the logistics transportation domain (Veloso, 1994).[1] In this situation there are three post offices A, B and C. In A there is a package p_1 and a truck. In B there is a package p_2. Suppose that two goals g_1 and g_2 are stated consisting in relocating p_1 and p_2 in C respectively. Any of the goals can be achieved first and then the other one. The arrows in Figure 4.2 depicts a path followed by the truck. In the corresponding plan, p_1 is loaded in the truck, the truck is moved from A to C (arc 1), where p_1 is dropped. Then, the truck is moved from C to B (arc 2), where p_2 is loaded. Finally, the truck is moved from B to C (arc 3), where p_2 is dropped. That is, g_1 is achieved before g_2 in this plan. The point is that any ordering restriction between the goals will eliminate some solutions. For example, if the ordering restriction $g_2 \prec g_1$ is given, the solution plan mentioned before does not belong to $PLAN_{\prec}(I,G)$.

Definition 4.3 (Valid Ordering Restriction,Extended Problem Description) *Given a problem description (I,G), an ordering restriction, $\prec$, on G is valid if $PLAN(I,G) = PLAN_{\prec}(I,G)$.*

An extended problem description is a triple $(I, G, \prec)$ where (I,G) is a problem description and $\prec$ is a valid ordering restriction.

Definition 4.3 says that the possible solutions are the same with or without considering $\prec$. Thus, no valid solution is lost if $\prec$ is considered. In the example presented before in the logistics transportation domain, no extended problem description can be stated because, otherwise, solutions will be lost. *This means that in the extended problem descriptions, $\prec$ has no semantical relevance relative to the problem.* However, $\prec$ has a significant operational relevance if used in an adequate way.

[1] In the logisticts transportration domain, a typical problem is, starting from a configuration from objects, locations and transportation means, to place the objects at different locations. There are differents sorts of locations and means of transportation. In addition, the means of transportation have certain operational restrictions. For example, a truck can only be moved between two places located within the same city (see Appendix B).

Example. The meaning of the ordering restrictions can be illustrated by retaking the example presented in Section 3.5. In this example some areas of the workpiece shown in Figure 4.3 are processed. The geometrical reasoner states several ordering restrictions including: $processed(H) \prec processedH1(U1)$. This ordering restriction states that the processing area H must be processed before the area U_1. This restriction is stated as a result of the following geometrical consideration: the area U_1 cannot be processed until the area H has been removed because the second one covers the first one. The condition stated in Definition 4.3 is illustrated in this example: even if the ordering restriction is not stated explicitly, any correct plan to manufacture the workpiece will process H before U_1. Thus, $processed(H) \prec processedH1(U_1)$ is a valid ordering restriction.

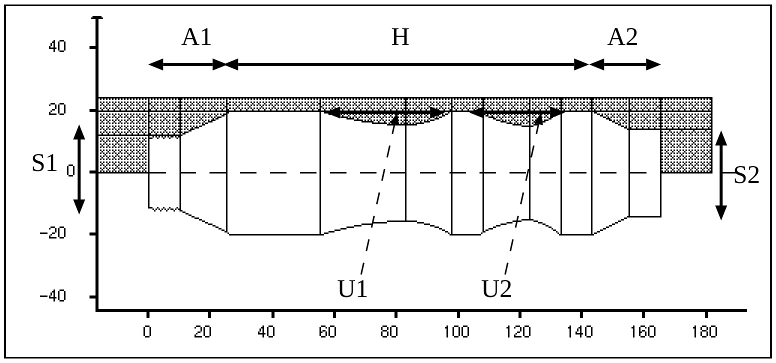


Figure 4.3: Display of a rotational symmetrical workpiece.

4.2 Dependencies between Goals

Before storing a solution plan as a new case, CAPLAN/CBC computes the order in which the goals were achieved in the plan. These ordering restrictions, called dependencies, are taken into account during retrieval. The name *dependency* is motivated by circumstances occuring in the domain of process planning.[2]

Definition 4.4 (Goal Dependencies) *Let g, h be two goals achieved by a complete partial-order plan $Pl = (S, \rightarrow, B)$. The goal g depends on the goal h, written $h <_P g$, if $establisher(h,P) <_\rightarrow establisher(g,P)$.*

In other words the goal dependencies reflect the order in which the goals are achieved in the plan. Figure 4.4 illustrates this definition. *step-i* and *step-j* correspond to *establisher(h,P)* and *establisher(g,P)* respectively. As *establisher(h,P)* $<_\rightarrow$ *establisher(g,P)* holds, we have that $h <_P g$ holds. That is, g depends on h.

[2]In the domain of process planning, processing the areas conforming a workpiece are the goals. Based on geometrical principles, some areas must always be processed after other areas and the number of alternatives to process an area decreases depending on the areas already processed.

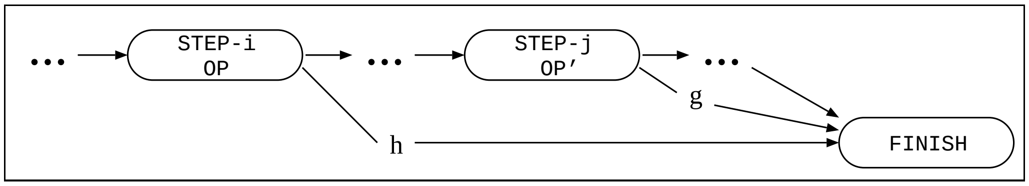


Figure 4.4: Dependencies between two goals.

Examples. Consider the examples in the domain of process planning and in the logistics transportation domain presented in the last section. In the first one, for any valid plan P manufacturing the workpiece depicted in Figure 4.3, the following dependency orders will hold: $processed(H) <_P processedH1(U_1)$ and $processed(H) <_P processedH2(U_1)$. In the second one, in the plan P depicted by the arrows in Figure 4.2, the dependency order $g_1 <_P g_2$ holds (g_1 corresponds to relocate p_1 in C and g_2 to relocate p_2 in C).

Computing the dependency order. Figure 4.5 shows the algorithm to compute the dependency order $<_P$ between the goals in G relative to the partial-order plan $P = (S, \rightarrow, \rightarrow_{CL}, B)$. The partial result is stored in $<_P$, which initially is assigned the empty set (line 1). The function *findPrecGoals(s,g)* finds all pairs of goals (g', g) such that g' is achieved by an step ordered preceding s (relative to $<_{\rightarrow}$). This function is called for each goal g in G and each step preceeding the establisher of g (step 2.2.1). Finally the transitive clousure of $<_P$ is returned (step 3). The partial result of the function *findPrecGoals(s',g)* is computed in $<_P$, which is initially assigned the empty set (step 1'). Then all the causal links producing g' and whose consumer is *finish* are checked to find if they are establisher of g' (steps 2', 2.1'). If this is this situation, $g' <_P g$ holds and correspondingly, the pair (g', g) is added in $<_P$ (step 2.1.1'). The recursive call is made with each step s'' immediately preceding s' (step 3'). Finally, the result is returned (step 4').

Algorithm 4.5 is illustrated with the plan depicted in Figure 4.6. In this plan four goals are achieved: g_1, g_2, g_3 and g_4. Their establishers are s_1, s_2, s_3 and *start* respectively. The last means that g_4 is established directly from the initial situation. In addition, $s_1 \rightarrow s_2$ and $s_1 \rightarrow s_3$ hold. As usual, all steps are ordered after *start* and before *finish*. Suppose that the first goal selected is $g = g_2$, thus $s = s_2$ and $s' = s1$. The function *findPrecGoals(s1,g)* returns $\{(g1, g_2), (g_4, g_2)\}$. The last one is obtained with the recursive call *findPrecGoals(start,g)*. The process with $g = g_1$ returns $\{(g_4, g_1)\}$, with $g = g_4$ returns $\{\}$ and with $g = g_3$ returns $\{(g_2, g_3), (g_1, g_3), (g_1, g_4)\}$. Finally, the result is the union of the four sets.

```
computeDependencyOrder($P, G$)
1 $<_P := \emptyset$
2 for-each $g \in G$
  2.1 s := establisher(g,Pl)
  2.2 for-each s' with $s' \to s$
    2.2.1 $<_P := <_P \bigcup findPrecGoals(s', g)$
3 return transitiveClousure($<_{Pl}$).
```

```
findPrecGoals($s', g$)
1' $<_{Pl} := \emptyset$
2' for-each $s''$ with $(s' \to g'@s'')$
  2.1' if (s" = finish and s' = establisher(g',P))
    2.1.1' $<_P := <_P \bigcup (g', g)$
3' for-each $s''$ with $(s'' \to s')$
  3.1 $<_{Pl} := <_{Pl} \bigcup findPrecGoals(s'', g)$
4' return $<_{Pl}$.
```

Figure 4.5: Algorithm to compute the dependency order $<_{Pl}$ between the goals in G relative to the partial-order plan $Pl = (S, \to, B)$.

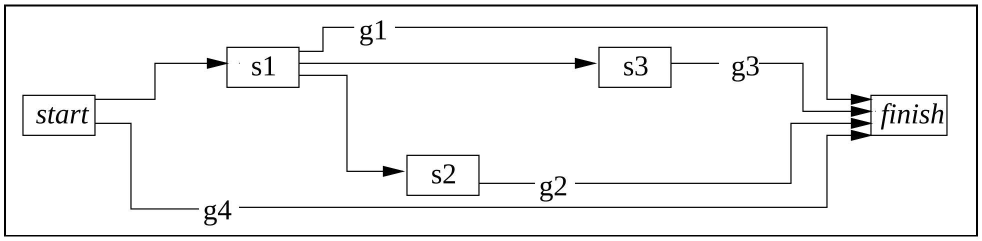


Figure 4.6: An abstract partial-order plan.

4.3 Retrieval Conditions

Given a case C, the dependency order among the goals G^C achieved in the plan contained in C is denoted by $<_C$. During retrieval the dependency order $<_C$ is compared with the ordering constraints of the problem. Two retrieval conditions have been studied in CAPLAN/CBC: the order inclusion condition and the order consistency condition. They differ on the degree of commitment that $<_C$ should have relative to $\prec$.

The order inclusion condition has a stronger commitment:

Definition 4.5 (Order Inclusion Condition) *A case C meets the ordering inclusion condition with respect to a problem $(I, G, \prec)$ if there is a substitution θ such that*

1. $G^C\theta \subset G$

2. *For every pair of goals $g, h \in G^C$, if $g\theta \prec h\theta$ holds, $g <_C h$ must also hold.*

Condition 1 is stated for two purposes: first, to avoid the possibility that a goal g in $G^C\theta - G$ interacts negatively with the goals in G or that the subplan achieving g interacts negatively with the subplan achieving G. Expressed in another way, g may add "noise" to the planning process because the planning effort to achieve the goals in G^C may increase as a result of achieving g. Second, it provides a simple criterium to discard candidate cases from the case base: cases achieving a number of goals greater than the number of goals of the problem are not considered. Further, these two purposes are directly related: the more additional goals (i.e., in $G^C\theta - G$) a case achieves, the more likely is it that negative interactions takes place. By forcing $G^C\theta \subset G$ to hold, no noise to the adaptation process can be caused by goals in $G^C\theta - G$ or by subplans achieving goals in $G^C\theta - G$.

Condition 1 is the typical requirement for case-based planners performing what we call goal-driven retrieval as it is solely based on the goals. Condition 2 expresses the dependency-directed retrieval technique by comparing $\prec$ against $<_C$. It states that $<_C$ must extend $\prec$ relative to the goals in $G^C\theta$. This condition ensures that for any two goals g, h, with $g\theta \prec h\theta$, $establisher(g, Pl^C) <_{\rightarrow} establisher(h, Pl^C)$ holds. Cases that meet this condition are more likely to be succesfully adapted to the new problem than cases that just match the goals of the problem. In a sense, the derivational path of cases meeting condition 2 is known to be compatible with the ordering restrictions of the goals.

Usually no case in the case base corresponds exactly to the solution of the problem. Instead, cases will partially match the new problem. This consideration is taken into account in Condition 1, where the goals of the case are required to match a subset of the goals of the problem (instead of all the goals). As it will be shown in Chapter 6, the same consideration is taken into account when comparing the initial states. Following this consideration, Condition 2 can be weakened to allow more cases to be considered. The idea is to discard from consideration cases that achieve their goals in an order that is not compatible with the orderings of the problem. That is, if the problem requires the condition $g\theta \prec h\theta$ to hold and in the case C, $h <_C g$ holds, then the incompatibility of the orders is an indication that there are parts of the case C that cannot be reused to solve the problem; namely, the parts establishing the ordering $establisher(h, Pl^C) <_{\rightarrow} establisher(g, Pl^C)$, where Pl^C is the solution plan in C. Formally the weakened condition can be formulated as follows:

Definition 4.6 (Order Consistency Condition) *A case C meets the ordering consistency condition with respect to a problem $(I, G, \prec)$ if there is a substitution θ such that*

1. *$G_C\theta \subset G$*

2. *For every pair of goals $g, h \in G_C$, if $g\theta \prec h\theta$ holds, $h <_C g$ must not hold.*

This means that the order consistency condition eliminates from consideration cases for which $establisher(h, Pl^C) <_{\rightarrow} establisher(g, Pl^C)$ holds. That is, cases for which the dependencies are incompatible with the ordering restrictions of the problem are eliminated from consideration. However, in contrast to the order inclusion condition, cases for which g and h are unordered relative to $<_C$ are still considered candidate cases. In this situation, it is possible that the case can be refined so that g is achieved before h as required.[3] In contrast, such a refinement is not possible for cases in which $h <_C g$ holds.

In Chapter 9, it will be shown that the order consistency condition effectively improves the accuracy of the retrieval phase and thus the performance of the overall case-based problem solving process.

4.4 Sequences of Dependency Classes

The order consistency condition eliminates from consideration cases having a dependency order incompatible with a given problem. As a result, cases meeting this condition are more likely to be adapted to the new problem with less difficulty than cases that just match some goals of the problem. In general, however, stating an effective retrieval condition is not useful unless an indexing structure that allows to test the condition efficiently is provided. In CAPLAN/CBC an indexing structure has been conceived and developed that allows to test the order consistency condition efficiently.

Once CAPLAN/CBC decides to store a solution plan Pl of the problem (I, G) as a new case C, several steps are performed: [4] first, the dependency order between the goals in G, $<_C$ is computed (see Figure 4.5). Then, the *sequence of dependency classes* is computed based on $<_{Pl}$. Finally, the case is indexed according to its sequence of dependency classes.

Definition 4.7 (Sequence of Dependency Classes) *Given the dependency order $<_C$ between goals in G, a sequence of dependency classes relative to $(G, <_C)$ is a sequence of sets $[G_1, ..., G_m]$ such that:*

1. *$\forall i \ (\emptyset \neq G_i \subset G)$ holds.*
2. *$G_i \bigcap G_j = \emptyset$ if $i \neq j$ holds.*
3. *$\bigcup_i G_i = G$ holds.*
4. *$\forall g \in G_i, h \in G_j \ (h <_C g)$ does not hold if $i < j$.*

[3] Whether this refinement is possible or not depends on the *context* and particularly on the initial features of the problem. As will be shown in Chapter 6, CAPLAN/CBC learns so that in succesive retrieval episodes the refinement of the retrieved cases to solve new problems is more likely to occur.

[4] The decision of whether to store an obtained solution, depends on the contribution of the retrieved cases to the problem solving effort (see Chapter 7).

5. *$\forall i \forall g, h \in G_i$ $(g <_C h)$ does not hold.*

In other words, a sequence of dependency classes relative to $(G, <_C)$ is a partition of G that is made based on $<_C$. The first three conditions ensure that $[G_1, ..., G_m]$ is a partition of G. Condition 4 states the connection between the sets in the partition and $<_C$. Notice that Condition 4 reflects Condition 2 of the order consistency condition (see Definition 4.6). This plays a key role in the retrieval phase as will be shown in the next section. Condition 5 ensures that the trivial partition (i.e., $[G]$) is not valid.

Example of dependency classes. Let $G = \{g_1, ..., g_6\}$ and suppose that $g_1 <_C g_2 <_C g_3$ and $g_4 <_C g_2 <_C g_5$ holds. In this example three sequences of dependency classes can be identified: $[\{g_1, g_4, \mathbf{g_6}\}, \{g_2\}, \{g_3, g_5\}]$, $[\{g_1, g_4\}, \{g_2, \mathbf{g_6}\}, \{g_3, g_5\}]$ and $[\{g_1, g_4\}, \{g_2\}, \{g_3, g_5, \mathbf{g_6}\}]$. That is, g_6 can be belong to any of the three dependency classes.

Algorithm for computing a canonical sequence of dependency classes. The previous example shows that there might be several partitions that meet the five conditions of Definition 4.7. However, the algorithm depicted in Figure 4.7 always finds the same partition for a given $(G, <_C)$, denoted as *the canonical sequence of dependency classes* relative to $(G, <_C)$. The idea is to take the set of minimal goals relative to $<_C$ (step 1). In the example, the minimal set is $\{g_1, g_4, g_6\}$. The function is called recursively with the remaining goals (step 3.1). The operator $\bullet$ performs a concatenation (i.e., $[G] \bullet [G'] = [G, G']$). No recursive call is made when the goals are exhausted (step 2). The result is returned in *Seq* (step 4). In the previous example the canonical sequence is $[\{g_1, g_4, g_6\}, \{g_2\}, \{g_3, g_5\}]$.

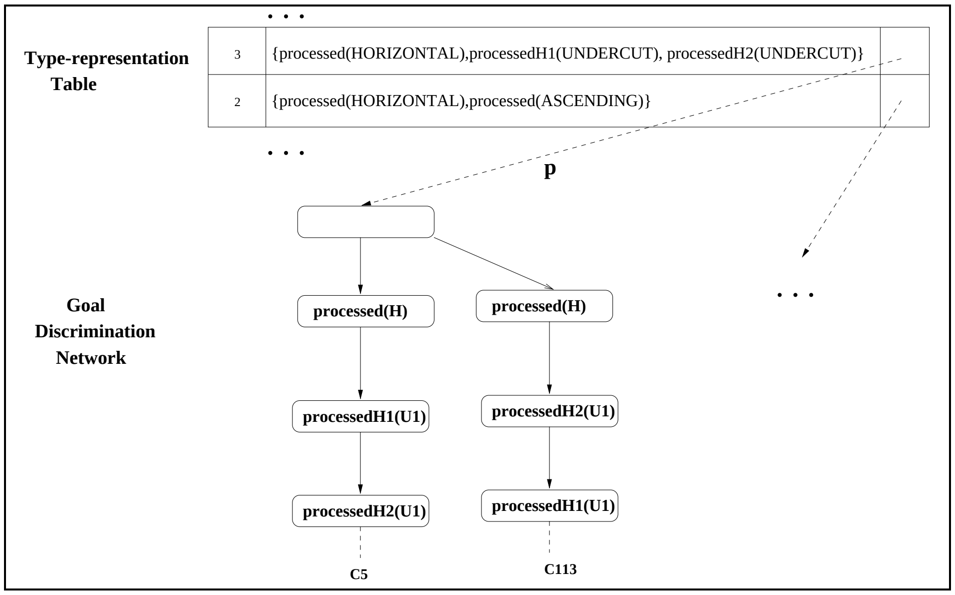
```
computeDependencyClasses($G, <_C$)
1 $G'$ := minimal(G,$<_C$)
2 if $G - G' = \emptyset$
  2.1 Seq := [G']
3 else
  3.1 Seq := [G']$\bullet$ computeDependencyClasses($G - G'$,$<_C$)
4 return Seq
```

Figure 4.7: Algorithm to compute a sequence of dependency classes relative to the pair of goal and dependencies $(G, <_C)$.

4.5 Indexing Structure of the Case Base

The sequence of dependency classes as obtained by the function *computeDependencyClasses*$(G, <_C)$ are used to construct the index structure in CAPLAN/CBC. The index structure consists of three levels: at the top level cases are discriminated by their sequences of dependency classes. At the intermediate level cases are discriminated by their initial features. Finally, all cases are listed at the bottom level.

In this section the construction of the top level is explained in detail.[5] Chapter 7 explains how the intermediate level is constructed. The top level contains two structures: the type-representation table and the goal discrimination network (*GDN*). The type-representation table is based on a similar table introduced in PRODIGY/ANALOGY. Each entry in this table has the form *(n,typ-rep,ptr)*. *n* indicates the number of goals in the type-representation *typ-rep*. The type-representation of a set of goals is obtained by replacing the arguments of the goals with their types. For example, continuing with the example of Section 3.5, the type-representation of {*processed(H), processedH1(U1), processedH2(U1)*} is {*processed(HORIZONTAL), processedH1(UNDERCUT), processedH2(UNDERCUT)*}. *ptr* is a pointer to a tree in the GDN. The principle for constructing this table is that *all cases achieving goals that have the same type-representation can be accessed below the pointer of the corresponding entry in the type-representation table.* The entry depicted in the type-representation table in Figure 4.8 corresponds to the case described in section 3.5.3. Thus, the case $C5$ can be accessed below the pointer p.



Type-representation Table
3
{processed(HORIZONTAL),processedH1(UNDERCUT), processedH2(UNDERCUT)}
2
{processed(HORIZONTAL),processed(ASCENDING)}
p
Goal Discrimination Network
processed(H)
processed(H)
processedH1(U1)
processedH2(U1)
processedH2(U1)
processedH1(U1)
C5
C113

Figure 4.8: Fragment of the first level of the indexing structure in CAPLAN/CBC.

The Goal Discrimination Network (GDN). The goal discrimination network (GDN) is a collection of trees, each of which is pointed to by a unique entry in the type-representation table. The trees are constructed based on the sequences of

[5]We are assuming that all goals in the solution interact (see Section 2.6).

dependency classes of the cases. The root of each three is a dummy node. Nodes different from the root contains goals so that *any path from any son of the root to any of its descendent leafs is a canonical sequence of dependency classes.* More concrete, if G_1 denotes the goals contained in a node, $node_1$, son of the root. G_2 denotes the goals contained in a node, $node_2$, son of $node_1$. By continuing in this way G_m denotes the leaf node, $node_m$, son of $node_{m-1}$. Then $[G_1, G_2, ..., G_m]$ is a canonical sequence of dependency classes. Further, *all canonical sequences of dependency classes relative to goals having the same type-representation are represented in the same tree of the GDN (the one pointed by the corresponding entry in the type-representation table).* Cases are pointed below the leaf node of the tree in the GDN such that the path from the root to this leaf node represents the canonical sequence of dependency classes relative to the goals achieved in the case and their dependency order. For example, the case $C5$ is pointed below the leaf node n because the canonical sequence of dependency classes of the goals achieved in $C5$, $[\{processed(H)\}, \{processedH1(U1)\}, \{processedH2(U1)\}]$, is represented in the path from the root r to n (excluding the root).

Algorithm for indexing in the GDN. Algorithm 4.9 indexes a new case C in the architecture of the case base. Its input are the case C, the goals G achieved by C, the canonical sequence of dependency classes Seq and the current index structure Idx. The algorithm begins by computing the type-representation of G (step 1). The function *findEntry(t-repG,Idx)* finds the entry of *t-repG* in the type-representation table if there is one. In this situation, the entry together with Idx are returned. If no such an entry exists, a new entry with *t-rep* is added. In this situation the new entry together with the modified index are returned (step 2). The case is indexed in the tree indicated in *entry* according to Seq (step 3). Finally, the modified index structure *Idx"* is returned (step 4).

```
indexCaseGDN($C, G, Seq, Idx$)
1 t-repG := type-representation(G)
2 (entry,Idx') := findEntry(t-repG,Idx)
3 Idx" := indexGDN(C,root(tree(entry)),Seq,Idx',1)
4 return Idx"
```

Figure 4.9: Algorithm to index a new case C according to its canonical sequence of dependency classes Seq. C achieves a set of goals G and Idx is the current indexing structure.

The function *findEntry(t-repG,Idx)*. This auxiliary function is shown in Figure 4.10. The variable *table* is assigned the type-representation table of *Idx* (step 1). If there is an entry in *table* containing the same type representation as *t-repG*, this entry is returned together with the unmodified index, Idx (step 2). Otherwise a new entry for *t-repG* is added to the type representation table of *Idx* (steps 3-6). The new entry and the modified Index, *Idx'* are returned. The function *number(entry)* returns the field *n* of *entry* and *typeRep(entry)* returns the field *typ-rep* of *entry*. In addition, *newEntry(t-repG)* creates a new entry such that *typeRep(entry) = t-repG*. The function *newTreeGDN()* creates a new tree consisting of a single node, the dummy node, and *addEntryTypeRepTable(entry,Idx)* adds *entry* as a new entry to the type-representation table of *Idx*, and returns the modified *Idx*.

```
findEntry(t-repG,Idx)
1 table := typeRepTable(Idx)
2 for-each entry $\in$ table with number(entry) = size(t-repG)
  2.1 if typeRep(entry) = t-repG
    2.1.1 return (entry,Idx)
3 entry := newEntry(t-repG)
4 tree := newTreeGDN()
5 assignPtr(entry,tree)
6 Idx' := addEntryTypeRepTable(entry,Idx)
7 return (entry,Idx')
```

Figure 4.10: Auxiliary function used in the algorithm to index a new case in the GDN, indexCaseGDN(C,G,Seq,Idx).

The function *indexGDN(C,node,Seq,Idx',i)*. This auxiliary function is shown in Figure 4.11 and is initially called with the parameter *node* instantiated with the root of the tree in which the case must be indexed and the value of the parameter i set to 1 (see step 3 of the algorithm *indexCaseGDN(C,G,Seq,Idx)*). i is a counter of *Seq*, which has the form $[G_1, ..., G_m]$ and *node* is a descendent of the root. i also indicates how many nodes are contained in the path from the root to *node*. Thus, if $i = 1$ holds, *node* is the root. The idea is to find if there is a child n_1 of the root containing goals matching G_1. If this is the situation, if there is a child of n_1 containing goals matching G_2 and so on. The process continues in this way until a node n_k is reached for which none of its children match G_{k+1}. In this situation the case is indexed below n_k. To do this, *setChildren* is assigned the set of children of *node* (step 1) and G' is assigned G_i (i.e., the i-th partition of *Seq*, step 2). Step 3 checks that $SetChildren$ is nonempty and that there is a child of *node*, $node'$, containing goals that match G_i. If this is the situation, the process is repeated recursively on $node'$ and $i + 1$ (step 3.3). If this is not the situation, either *node* is a leaf or no child of *node* contains goals matching G_i. In the second case, $G_i, ..., G_m$ are each stored in a different node (step 4.3.1). The case is indexed below the node containing G_m (step 4.4).[6] A special situation occurs if *node* is a leaf. This means that $i = m+1$ holds. In

[6]The case must still be indexed acording to its initial state at the intermediate level of the architecture (see Chapter 7).

this situation step 4.3.1 is skipped and the case is indexed below *node*. The function $match(G', G'', \theta)$ returns true if *G'* matches *G"*. In this situation θ is instantiated with the corresponding substitution. The calls *apply(θ,G)* and *apply(θ,C)* return the result of applying the substitution θ to the set of goals G and the case C respectively.

```
indexGDN(C,node,Seq,Idx',i)
1 setChildren := children(node)
2 G' := Seq(i)
3 if (setChildren $\neq$ {}) and ($\exists$ node' $\in$ setChildren with match(G',goals(node'),$\theta$))
  3.1 Seq' := apply($\theta$,Seq)
  3.2 C' := apply($\theta$,C)
  3.3 Idx" := indexGDN(C',node',Seq',Idx',i+1)
4' else
  4.1 n := size(Seq)
  4.2 currentNode := node
  4.3' if i $\leq$ n
    4.3.1 for-each j $\in \{i, ..n\}$
      4.3.1.1 node' := newNode(G')
      4.3.1.2 assignChild(currentNode,node')
      4.3.1.3 currentNode := node'
      4.3.1.4 G' := Seq(j)
  4.4 Idx" := indexCaseBelow(C,currentNode,Idx')
5 return Idx"
```

Figure 4.11: Auxiliary function used in the algorithm to index a new case in the GDN, indexCaseGDN(C,G,Seq,Idx).

4.6 Efficient and Accurate Retrieval

As discussed in Section 4.3, given a new problem, retrieved cases that meet the order consistency condition are more likely to be effectively reused than retrieved cases that just match some goals of the problem. In this section will be shown how the order consistency condition can be tested by using the GDN instead of testing it on each case in the case base.

Consider two nodes n_1 and n_2 in a tree in the GDN, such that n_1 precedes n_2 and neither of them is the root of the tree. If *goals(n)* denote the goals stored in a node n in the GDN, then *goals(n_1)* and *goals(n_2)* are in canonical sequences of dependency classes relative to a dependency order $<_C$ for any case C indexed below n_2. Further, the class of goals *goals(n_1)* occurs before the class of goals *goals(n_2)* in these sequences. Thus, if $g \in$ *goals(n_1)* and $h \in$ *goals(n_2)* hold, $h <_C g$ does not hold in any case C below n_2. For example, in the situation depicted in Figure 4.8, *processedH1(U1)* $<_{C5}$ *processed(H)* does not hold. This illustrates the following property, called the *GDN-property*:

> Given an order restriction $g \prec h$ in a new problem, a substitution θ and two nodes n_1 and n_2 in a tree in the GDN, if $g = g'\theta \in goals(n_1)$ and $h = h'\theta \in goals(n_2)$ hold and n_1 is an ancestor of n_2, then $h'\theta <_C g'\theta$ does not hold for any case indexed below n_2.

Figure 4.12 illustrates this property. g and h match g' and h' respectively. Since n_1 is a predecessor of n_2 in a tree in the GDN, either $g' <_C h'$ holds or g' and h' are unordered modulo $<_C$ (Figure 4.12 (b)). This property is crucial to evaluate the order consistency condition efficiently; instead of testing this condition case by case, it can be tested by matching goals contained in nodes in the GDN; as the paths in the trees in the GDN represent sequences of dependency classes for all cases indexed below them, traversing these trees is equivalent to test the order consistency condition on several cases simultaneously.

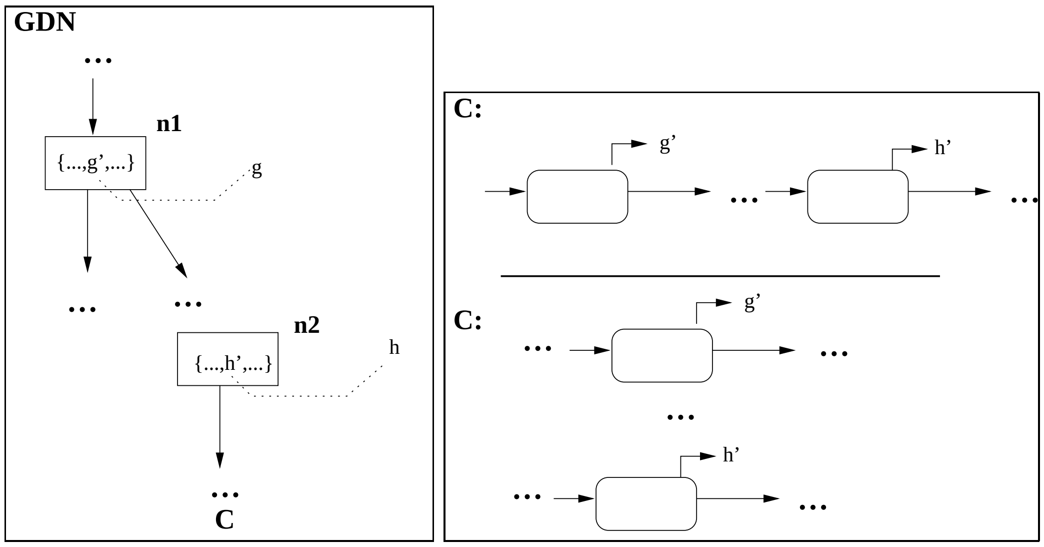


Figure 4.12: Illustration of a property of the GDN. (a) Part of a tree in the GDN. (b) Two possible plan fragments in C.

The GDN-property can be used to test if a case C meets the order consistency condition relative to the ordering restriction $g \prec h$: let C be indexed below the path $rt, n_1, ..., n_m$, where rt is the root of a tree in the GDN and n_m is a leaf. Thus, $[goals(n_1), ..., goals(n_m)]$ is the canonical sequence of dependency classes relative to the goals achieved in C and to $<_C$. In this context, the following possibilities can be identified:

- If there is a substitution θ such that $g \in goals(n_i)\theta, h \in goals(n_j)\theta$ and $i \leq j$ hold, the order consistency condition is met. If $i > j$ hold, the order consistency condition is not met.

- if no substitution θ exists such that $g \in goals(n_i)\theta$ and $h \in goals(n_j)\theta$ hold, the ordering consistency condition is trivially met.

In resume, if there is a substitution θ such that $g \in goals(n_i)\theta, h \in goals(n_j)\theta$ and $i > j$ hold, the order consistency condition relative to $g \prec h$ is not met. Otherwise, it is met. In other words, *the only way that the order consistency condition is not met relative to an ordering $g \prec h$ in the current problem is if g is matched by a goal in a node n_g, h is matched by a goal in a node n_h and n_h is a predecessor of n_g.* The algorithm that follows examines this condition when traversing the GDN.

```
retrieveCandidateCasesGDN(G,≺,Idx)
1 i := size(G)
2 t-repG := typeRep(G)
3 table := typeRepTable(Idx)
while 4. i > 0
  4.1 setEntries := allEntriesWithNumber(Table,i)
  4.2 for-each entry ∈ setEntries
    4.2.1 if typeRep(entry) ⊂ t-repG
      4.2.1.1 (cand,G') := matchingCases(G,≺,root(ptr(entry)),∅,G)
      4.2.1.2 if cand ≠ ∅
        4.2.1.2.1 return (cand,G')
  4.3 i := i - 1
5 return (∅, ∅)
```

Figure 4.13: Algorithm to retrieve a set of candidate cases from the indexing structure Idx and for a given Problem $(I, G, \prec)$.

Algorithm for retrieving candidate cases. The algorithm to retrieve a set of candidate cases is shown in Figure 4.13. The input are the set of goals G, the ordering restrictions between the goals $\prec$ and the indexing structure Idx. The output is a set of cases meeting the order consistency condition, with each case matching the same subset, G', of G.[7] G' is also returned. The algorithm will try to find cases covering all goals in G. If this fails, it will try to find cases covering all goals but one. It continues in this way, finally trying to find cases covering a single goal. If none is found, the empty set is returned. The counter i indicates how many goals the algorithm tries to cover. Thus, initially i is the number of goals in G (step 1). *t-repG* and *table* are assigned the type-representation of G and the type-representation table in *Idx* respectively (steps 2 and 3). For each value of i, the following steps are made: first, all entries in *table* having i goals in their type-representation item are collected in *setEntries* (step 4.1). Then, each entry in *setEntries* is tested to check if the

[7]A further selection takes place by comparing the initial states of the selected cases and the problem as will be described in chapter 6.

type representation of the entry is equal to a subset of the type representation of G (step 4.2.1) and if there is a collection, *cand*, of cases such that their dependencies meet the ordering consistency condition relative to a subset of G, G' (step 4.2.2). If this is the situation, *cand* and G' are returned. If all possible values of i (i.e., $\{size(G), size(G)-1, ..., 1\}$ are exhausted and no such a collection has been found, the empty set is returned (i.e., no cases are retrieved, step 5).

```
matchingCases($G$,$\prec$,$node$,$G^{Match}$,$G^{NotMatch}$)
1 setChildren := children(node)
2 for-each node' $\in$ setChildren
  2.1 $G^{NewMatch} := goals(node') \bigcap_{\theta} G^{NotMatch}$
  2.2 if size(goals(node')) = size($G^{NewMatch}$)
    2.2.1 if testOrderings(G,$\prec$,$G^{Match}$,$G^{NewMatch}$)
      2.2.1.1 (cand,$G^{NextMatch}$) :=
                  matchingCases(G,$\prec$,node',$G^{Match} \bigcup G^{NewMatch}$, $G^{NotMatch} - G^{NewMatch}$)
      2.2.1.2 if $cand \neq \emptyset$
        2.2.1.2.1 return (cand, $G^{Match} \bigcup G^{NewMatch} \bigcup G^{NextMatch}$)
3 return $(\emptyset, \emptyset)$
```

```
testOrderings($G$,$\prec$,$G^{Match}$,$G^{NewMatch}$)
1' for-each g, h $\in$ G with $g \prec h$
  1.1' if ($h \in G^{Match}$ and $g \in G^{NewMatch}$)
    1.1.1' return FALSE
2' return TRUE
```

Figure 4.14: Auxiliary function *matchingCases(G,$\prec$,node,$G^{NotMatch}$,G^{Match})* called in the algorithm to retrieve a set of candidate cases, retrieveCandidateCasesGDN(G,$\prec$,Idx).

The auxiliary function *matchingCases*(G,$\prec$,node,G^{Match},$G^{NotMatch}$). This function, which is shown in Figure 4.14, is based on the GDN-Property; if there is a substitution θ such that $g\theta \in goals(n_i), h\theta \in goals(n_j)$ and $i > j$ hold, the order consistency condition relative to $g \prec h$ is not met. In any other situation, the condition is met. *node* indicates the last node that has been examined. G^{Match} indicates the goals in G that have been matched by a goal in the path from the root of the tree pointed by *entry* to *node*, and $G^{NotMatch}$ contains the nonmatched goals in G. As a result, initially *node* is the root of the tree, G^{Match} is the empty set and $G^{NotMatch}$ is G (see step 4.2.1.1 of Figure 4.13). *node'* is the child of *node* that is been currently examined (step 2). If there is a subset, $G^{NewMatch}$, of $G^{NotMatch}$

matching *goals(node')*, the order consistency condition is tested relative to all ordering restrictions $\prec$ and to G^{Match} and $G^{NewMatch}$. This is done by the function testOrderings(G,$\prec$,G^{Match},$G^{NewMatch}$) called in step 2.2.1 and shown also in Figure 4.14. If the test is positive the function *matchingCases* is recursively called (step 2.2.1.1). If the recursive call returns a nonempty set of candidate cases *cand*, *cand* is returned (step 2.2.1.2.1). In addition, the subset of G achieved by the cases in *cand* is computed by joining G', the subset of G returned by the recursive call, and $G^{NewMatch}$, the subset of G matching *goals(node')*. If all children of *node* has been tested without success, a pair of empty sets is returned (i.e., no candidate cases are retrieved, step 5).

4.7 Multi-Case Retrieval

To retrieve multiple cases a top-down strategy is followed: first, a call to the function *retrieveCandidateCasesGDN(G,$\prec$,Idx)* is made with the parameter G instantiated with all the goals of the new problem (see Figure 4.13). This function returns a set of candidate cases *cand*, each covering the same subset of G, G', which is also returned. If *cand* is empty, there are no cases achieving any subset of G and meeting the order consistency condition. Otherwise, the function is called again with the remaining goals $G - G'$. The process continues until all goals in G have been exhausted. At the end a sequence of the form $[(CC_1, G_1), ..., (CC_m, G_m)]$ is obtained such that

1. CC_i is a set of cases achieving G_i, a subset of G.
2. $G_i \bigcap G_j = \emptyset$ holds if $i \neq j$ holds and $\bigcup_i G_i = G$ holds.
3. CC_m may be empty.

A final selection on which case in CC_i to retrieve is made by comparing the initial states as will be shown in Chapter 6. Statement 2 says that $[G_1, ..., G_m]$ is a partition of G. Statement 3 says that the last set of goals may not be covered by any case. This is a typical situation as it cannot be expected that all goals can be covered. In chapter 8 a method for merging cases is presented and its effectiveness is evaluated.

4.8 Discussion

The core of the dependency-driven retrieval technique is that problems are given in the form of *extended problem descriptions* $(I, G, \prec)$. An extended problem description together with the order consistency retrieval condition defines an intentional description of a collection of plan fragments $Coll$; namely, a plan fragment $P = (S, \rightarrow, rightarrow_{CL}, B)$ is in $Coll$ if for every ordering restriction $g \prec h$, $h <_P g$ does not hold. That is, $establisher(h, P) <_\rightarrow establisher(g, P)$ does not hold. Under this perspective, the retrieval problem can be re-stated as to find a case C in the case base such that the solution of C contains a plan fragment P' in *Coll* which can be extended to a complete solution by reusing C.

Clearly, dependency-driven retrieval will usually result in the retrieval of more appropriate cases than goal-driven retrieval as the former is more informed (i.e., goals of the candidate cases must not only match the goals of the problem but their dependencies must be compatible with the ordering restrictions of the problem). It could be, however, that the performance of the retrieval decreases and as a result the performance gains obtained by reusing the more appropriate cases could be lost. However, the time costs of performing dependency-driven retrieval is usually not greater than goal-driven retrieval. In fact, as will be shown in Chapter 9 the former tend to be smaller than the latter. An explanation of this can be made with an the use of combinatorics: when performing goal-driven retrieval, sets of goals are matched (the goals of the problem and of the cases). To simplify the discussion, suppose that two sets of size n are matched. At the worst case, $n!$ permutations of goals are made. When performing dependency-driven retrieval, partially ordered sets of goals are matched (i.e., the dependency order of the cases against the ordering restrictions of the problem). The number of permutations that in the worst case have to be performed, decreases with the number of ordering restrictions and dependencies. In a limit situation, if both sets are totally ordered, a single permutation of goals needs to be considered.

As we saw the GDN allows to test the order consistency condition for several cases simultaneously instead of one by one. As will be shown in the experiments (see Chapter 9), this will result in significant improvements in retrieval time for domains such as process planning in which several ordering restrictions can be predefined. However, the same experiments show that when fewer or none ordering restrictions are given the GDN should not be used because the retrieval time is increased in a significant way. The reason for the increase is that in any tree T of the GDN the same set of goals is listed several times; one for each path from the root of T to a leaf. By traversing these trees in the way described by algorithm *retrieveCandidateCasesGDN(G,$\prec$,Idx)* (see Figure 4.13), the same set of goals will be matched several times. In contrast, the more ordering restrictions are given, the less nodes are visited in a tree. In the domain of process planning, for example, the number of ordering restrictions typically increases with the number of goals. In the worst case, five goals can be stated without any ordering restrictions; namely, the goals corresponding to manufacturing the five outlines (see Appendix A). However, any additional goal, which corresponds to the machining of a feature covered by an outline, necessarily add at least one ordering restriction. Thus, ten goals will contain at least five ordering restrictions.

An issue that remains to be discussed is how the characteristics of the domain theory affect the retrieval procedure. In Chapter 7 the issue regarding the relation of dependency-driven retrieval and the characteristics of the domain will be discussed.

Chapter 5

Adaptation of Cases with Complete Decision Replay

Adaptation based on replay is widely used in case-based planners searching in the space of states and in the space of plans (Veloso and Carbonell, 1993; Bhansali and Harandi, 1994; Blumenthal and Polster, 1994; Ihrig and Kambhampati, 1994). In this approach, the *derivational trace* of the retrieved cases is reconstructed relative to the current problem (see Section 2.6). Different factors affect the performance of this method such as the the search space of the base-level planner, whether case-based and first-principles planning interleave and the degree of repair (Muñoz-Avila and Weberskirch, 1996a).

As originally proposed in PRODIGY/ANALOGY, the derivational path was anotated with failure reasons in a validation structure (Veloso, 1994). These failure reasons expressed situations proper of state-space planners such as the situations encountered by the state-space planner PRODIGY (Blythe et al., 1992). Later implementations in plan-space planners considered only the derivational path but not any failures that occurred during the adaptation process (e.g., (Ihrig and Kambhampati, 1994)). Thus, they have a *low degree of repair* as the repair effort is totally left to the first-principle planner. A direct consequence of this flaw is that a large amount of planning effort may be needed to complete the solution of the new problem because failures occuring during the solution of the cases are not taken into account.

In this chapter, we introduce a new adaptation method based on replay that we called *Complete Decision Replay.* This method is conceived on a plan-space planner and improves previous approaches using replay based on these planners by increasing the degree of repair (Muñoz-Avila and Weberskirch, 1996b). Moreover, we will see that complete decision replay enables the user to interact during the adaptation process.

5.1 Motivation of Complete Decision Replay

In this thesis we combine static retrieval techniques (see the previous chapter) with dynamic retrieval techniques (which will be discussed in the next chapter). However, independent of the retrieval technique used, it is not realistic to suppose that adequate cases will always be found. That is, cases that perfectly fit into a solution plan of the new problem. There are two reasons for this:

- There are no cases in the case base that perfectly "fit" into a solution plan of the new problem.
- Retrieval cannot take much time. Otherwise, the trade-off between retrieval and reuse effort is lost (Veloso, 1994; Francis and Ram, 1995a).

The first reason is a recurrent argument in CBR: given that the space of problems is usually very large, it is not feasible to suppose that there is a case that solves exactly the same problem. The meaning of a case "fitting" into the new problem can be precisely defined for CBP; namely, if the solution plan in the case is a *subplan* of a solution plan of the new problem. Here is where the second argument is considered: even if such a case exists it may take too much time to find it as it would require to test for every candidate case if its solution plan can be extended to a solution plan of the new problem.

Abstract Example. Figure 5.1 compares complete decision replay with standard replay using an abstract example. The continuous lines represent the derivational path driving to the solution plan of the case. In addition, the failed exploration attempts have been also depicted (the discontinuous lines). The case has been selected for replay and we are supposing that the decision labeled B cannot be replayed in the new situation. Further, we are supposing that the decision labeled A needs also to be rejected to complete the solution of the new problem. Solution (a) sketches the search path that must be followed when completing the plan obtained with replay. Notice that during completion of the new solution, some of the failed attempts made in the case are repeated, for example decisions labeled C and D. In complete decision replay, justifications of failed attempts are constructed as part of the cases. These justifications are reconstructed as part of the adaptation process. As a result, performing unnecessary backtracking is avoided during the completion process (see solution (b)).

We will now explain how complete decision replay was conceived and implemented in CAPLAN/CBC, which is based on the plan-space planner CAPLAN. In particular, we will explain how the justifications are made and how they are reconstructed during replay. But, before, we will make an overview of the aspects of CAPLAN which are relevant for the adaptation process.

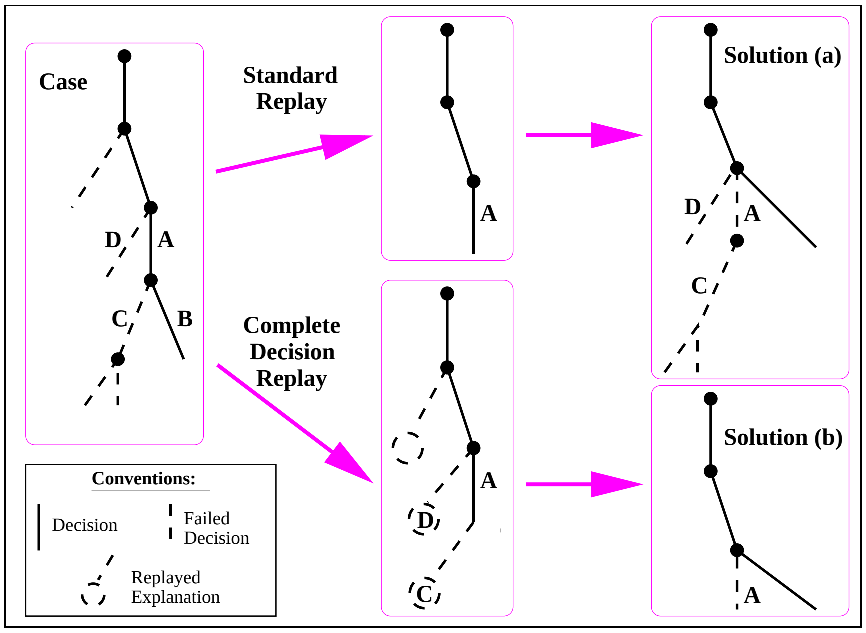


Figure 5.1: Replay of a case and completion after (a) standard replay and (b) complete decision replay.

5.2 The Base-Level Planner CAPLAN

CAPLAN is a plan-space planner based on SNLP (the reader not familiar with SNLP is advised to read Section 2.3). There are several extensions made in CAPLAN with respect to SNLP:[1]

- Type information can be explicitly specified in the domain description.
- A dependency maintenance system has been incorporated.

The first aspect is related to the fact that as originally defined, SNLP considered only codesignation and noncodesignation constraints (see Section 2.1). CAPLAN implements SNLP but also handles type information explicitly. The second aspect is the most significant contribution of CAPLAN and plays a key role to perform complete decision replay.

REDUX. The dependency maintenance system incorporated is REDUX (Petrie, 1991a; Petrie, 1991b). By incorporating the generic REDUX architecture, CAPLAN is able to represent knowledge about plans and contingencies that occur during planning. Key concepts of REDUX are goals, constraints, and contingencies. Planning

[1]In this section an overview of some aspects of CAPLAN is made in order to explain how complete decision replay was conceived. A detailed description of CAPLAN can be found in (Weberskirch, 1995).

proceeds by applying operators to goals, which may result in subgoals and in assignments (Figure 5.2.a). Applying an operator is called a *decision* and represents a

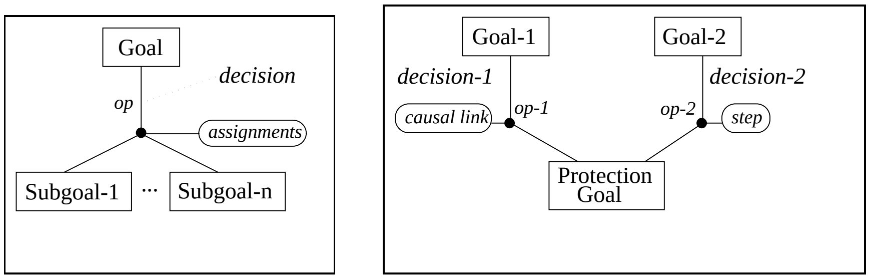


Figure 5.2: Representation of a (a) decision and (b) a threat in the subgoal graph.

backtracking point as different operators might be applicable to a goal.

Note about the term "applicable". In the parlance of REDUX and *thus of* CAPLAN, an operator is said to be applicable to achieve a goal if one of its effects in the add-list matches the goal modulo the bindings B of the plan and the constraints of the operator. Whether this operator is in fact applicable in the sense of Definition 2.2, can only be stated if subgoals of the decision corresponding to apply the operator to achieve the goal are also achieved.

Assignments in Planning. Assignments originally are thought to assign values to variables. More generally, they stand for modifications made in the plan. In SNLP, four possible modifications can be made to a partial-order plan $< S, \rightarrow, \rightarrow_{CL}, B >$ (thus, each of these modifications is represented as an assignment in REDUX):

- Addition of a plan step to S. New steps are added to establish preconditions.
- Addition of a causal link, $s \rightarrow p@s'$, to $\rightarrow$ and $\rightarrow_{CL}$. Causal links are added to establish preconditions (i.e., $p@s'$ denotes the precondition p of s'). The source of the causal link (i.e., s') is either a new step or an existing step in the plan. The former is called an establishment with a new plan step whereas the latter is called a simple establishment.
- Addition of a protection link, $s \rightarrow s'$, to $\rightarrow$. Protection links are added to solve threats.
- Addition of binding constraints to B. Binding constraints are added either to solve threats (i.e., to perform "separation") or when an operator is applied, in which case the constraints of the operator are added to B.

The mapping of SNLP concepts to REDUX concepts is straightforward: SNLP goals are mapped to REDUX goals and SNLP operators to REDUX operators. There are two types of SNLP goals:

- **Precondition (Domain) goals**. These are the goals corresponding to establish a precondition $p@s$ of a plan step s.
- **Planning (Protection) goals**. These are goals corresponding to the solution of a threat $s_3 \overset{+/-}{\longleftrightarrow} (s_1 \rightarrow p@s_2)$.

Correspondingly, SNLP operators are of two types:

- **Domain Operators**. These operators are used to achieve precondition goals. Thus, they are divided in two kinds: simple establishments and establishes with new plan steps. That is, in the former a causal link is added to the plan whereas in the latter a causal link and a new step are added to the plan.
- **Planning (Protection) Operators**. These operators serve to achieve planning goals. That is, with these operators threats, $s_3 \overset{+/-}{\longleftrightarrow} (s_1 \rightarrow p@s_2)$, are solved. Thus, there are three kinds of them:
  - Separation operators: binding constraints are added so that the effect p' of s_3, which is in conflict with the precondition $p@s_2$, cannot unify.
  - Promotion operators: the ordering constraint $s_2 \rightarrow s_3$ is added to solve the threat.
  - Demotion operators: the ordering constraint $s_3 \rightarrow s_1$ is added to solve the threat.

The Subgoal Graph. Goals and subgoals build the *subgoal graph*. It represents basic dependencies between goals and subgoal as well as between subgoals and decisions. Originally, REDUX makes the assumption that each goal can only have one parent goal. So the subgoal graph in fact is a tree. This is not adequate for SNLP and has been modified in CAPLAN for the following reason: protection goals represent threats and depend on two other goals (Figure 5.2.b). First, the goal with the decision that added the threatened causal link. Second, the goal with the decision that added the threatening step. This extension of the dependency structure is important for automatically identifying threats and threat resolutions that are no longer valid after the rejection of a decision.

Dependencies. Basically, REDUX represents validity and local optimality of decisions and dependencies among them (cf. (Petrie, 1992)). Important dependencies for CAPLAN are:

1. Subgoals depend on goals.
2. Subgoals depend on the decision that added them.
3. Decisions depend on the goal they are applied to.

4. Assignments depend on the decision that added them.

These dependencies are explicitly represented in the subgoal graph (see Figure 5.2); for example, the first dependency means that *Subgoal-1*, ..., *Subgoal-n* depend on *Goal*. These dependencies are also used to determine the validity of decisions, goals and assignments. For example, a subgoal is valid if its corresponding goal is valid (dependency 1) and the decision that added it is valid (dependency 2). In the example, *Subgoal-1* is valid if *Goal* and *decision* are both valid. As a result, *if a decision is rejected, the subgoal structure is traversed to reject depending decisions as well.* If, for example, *Goal* is rejected by the user, CAPLAN declares *Subgoal-1*, ..., *Subgoal-n*, *decision* and *assignments* as rejected.

The constraint system in CAPLAN computes connected components in a graph where the nodes contain the variables and the arcs are the constraints. If, for example, a decision is taken corresponding to the application of an operator that has the constraint $x = y$, this constraint is added as an edge between the node containing x and the node containing y. The constraint $x = y$ is also represented in an assignment of the decision.

5.3 Justifications of Decisions

As we saw in the previous section, every time a decision is made, new assignments are created representing the changes in the plan that take place as a result of making the decision. Assignments are the base on which justifications are made. A *justification* is a set of assignments, $\{a_1, ..., a_n\}$. For each assignment a_i it is possible to identify the decision D_i that added the assignment. CAPLAN constructs the *justifications* of every decision made (valid or rejected). Justifications play a key role to determine if the state of validity of a decision remains unchanged when another decision is rejected. If a decision D is rejected, its corresponding assignments A_D are removed. This affects any other decision D' in the following way:

- If D' is rejected and at least one assignment a_i in its justifications belong to A_D, then the decision is no longer rejected. That is, the problem solver may make D' later on.

- If D' is valid and at least one assignment a_i in its justifications belong to A_D, then the decision must be *retracted*. That is, the subgoal g that was achieved with D' becomes unachieved. Thus, the problem solver will need to achieve g again. If a decision is "retracted", it does not mean that the decision is rejected. Rejected decisions are known to have failed. Retracted decisions were previously known to be valid but their validity can no longer be guaranteed due to changes in the plan.

If a decision is rejected the subgoal graph is traversed to examine the validity of the dependent decisions.

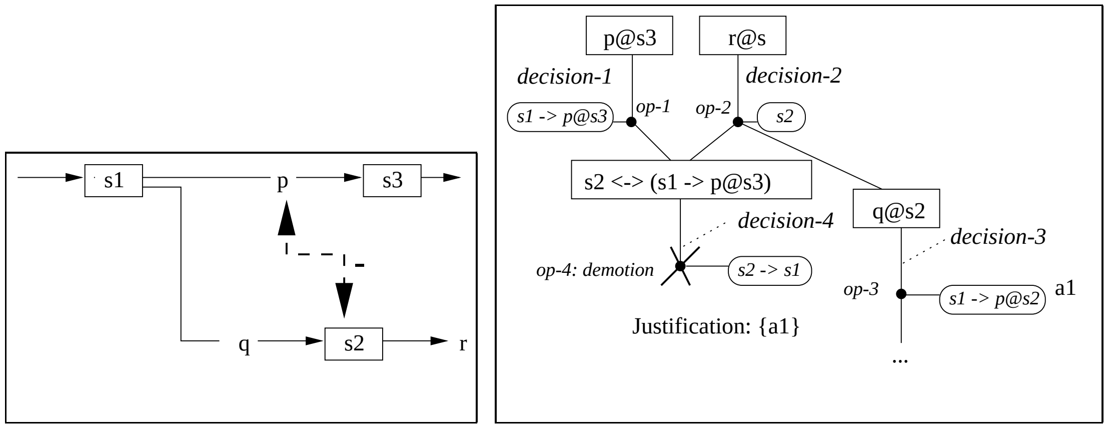


Figure 5.3: (a) Plan fragment illustrating a situation in which a threat occurs but demotion is not applicable to solve it and (b) part of the subgoal graph in which the plan fragment is represented.

Example of a Justification. An example of a justification is illustrated in Figure 5.3. A plan fragment is shown consisting of three plan steps and two causal links (part (a)). In addition, the threat $s_2 \overset{-}{\longleftrightarrow} (s_1 \rightarrow p@s_3)$ is depicted (the double arrow). The part of the subgoal graph representing this plan fragment is also depicted (part (b)). Goals are represented with boxes and assignments with rounded boxes (not all assignments associated with the plan fragment are shown). Three domain goals, $p@s_3$, $r@s$ and $q@s_2$ are represented together with the planning goal $s_2 \overset{-}{\longleftrightarrow} (s_1 \rightarrow p@s_3)$. A failure occurs if the planner pursues to perform a demotion (i.e., to add the ordering constraint $s_2 \rightarrow s_1$) to achieve the planning goal. The reason of the failure is that $s_1 \rightarrow p@s_2$ is a link in the plan and, thus, a cycle occurs in the plan if demotion is performed. This link is represented by the assignment a_1, which was introduced by *decision-3*. *decision-3* was made to achieve the subgoal $q@s_2$ and corresponds to an establishment with s_1. *decision-4* corresponds to the demotion operation and it is rejected. The justification of this rejection is $\{a_1\}$. If *decision-3* is rejected later on, *decision-4* is no longer considered rejected because tha assignment a_1 no longer exists. In this situation, *decision-4* may be made by the planner.

5.4 Contents of Cases

The subgoal graphs are part of the cases. The plan represented in a subgoal graph can be easily reconstructed by following the assignments of the valid decisions. The reason for this is that assignments indicate changes made in the plan (see the previous sections). Additionally, cases contain information concerning justifications of the decisions. The rationale behind storing the justifications as part of the cases is that they indicate the reasons for the state of validity of a decision.

> **Claim 5.1 (Justification Reconstruction Claim)** *If a decision is known from the case to be rejected and its justification can be reconstructed relative to the new problem, then the decision must also be rejected in the new situation.*

Latter in this chapter it will be explained why this statement, which we called the justifications-reconstruct claim is true. In particular it will be explained what is contained in the justifications for rejected decisions in SNLP. Before continuing a definition is introduced:

Definition 5.1 (Conflict Set) *Given a goal g the conflict set CS of g is the set of all applicable operators to achieve the goal.*

For each applicable operator to achieve a goal there is one and only one valid decision in the goal subgraph, which represents the application of the operator. Thus, we also use the term conflict set to refer to the set of decisions that can be made to achieve a goal.

The part of a case in CAPlan/CbC that is used for adaptation consists of the following elements:

- The subgoal graph.
- The conflict set of each goal and the decision made to achieve it. This decision is labeled as valid.
- The justifications of each rejected decision. These decisions are labeled as rejected.

Example of Information Stored in a Case. Continuing with the example discussed in the previous section, if the planning episode is to be stored as a case, CAPlan/CbC stores the subgoal graph depicted in Figure 5.3, all conflict sets and the justifications of the rejected decisions. In particular, *decision-4*, the fact that it is rejected and its justification (i.e., $\{a_1\}$) are stored in the case.

5.5 Complete Decision Replay

The idea of complete decision replay is to reconstruct the justifications stored in the cases relative to the current problem. In a sense, by making this reconstruction, the complete problem solving episode stored in the case is taken into account to solve the new problem. Complete decision replay is performed in three phases:

1. Reconstruction of the subgoal graph of the case with respect to the current problem.

2. Reconstruction of the justifications of the rejected decisions.

3. Completion of the partial solution obtained after 2 by the base-level planner CAPlan.

As explained in the previous chapter, once retrieval is made one or more cases are retrieved. Each retrieved case achieves a set of goals that match a subset of goals of the new problem (see condition 1 of the order consistent condition in Definition 4.6). That is, the goals of the retrieved case C match a subset G of goals of the new problem. The algorithm sketched in Figure 5.4 replays the decisions of the case C for the goals G. Pl is the initial plan (i.e., a plan representing the problem) and N is an initially empty list of goals that could not be matched immediately and so are delayed. Goals are delayed in the following situations:

- The goal is a domain goal and is achieved in the case by a simple establishment with s_1 and s_1 does not form part of the plan. As the order in which the plan stored in the case was generated is not necessarily the same as the order in which the plan to solve the new problem is generated, it is possible that at the moment where the simple establishment operator is to be replayed, the decision adding s_1 to the plan has not been replayed. In this situation, the simple establishment cannot be made yet.

- The goal is a threat $s_3 \overset{+/-}{\longleftrightarrow} (s_1 \rightarrow p@s_2)$ (i.e., a planning goal) and either s_3 or $s_1 \rightarrow p@s_2$ does not form part of the replayed plan yet. In this situation, the threat does not occur in the new situation yet.

Notice that there is no guarantee that the delayed step s_1 in the first situation will be replayed. In general, some decisions made in the case cannot be reconstructed. This occurs because their corresponding operators are not applicable relative to the new situation. An operator in not applicable when no binding of its variables exists that is consistent relative to its binding constraints and the binding constraints B of the current plan. The same can happens for s_1, s_2 or s_3 in the second situation.

Phase 1 (**ReplaySubgoalGraph**) reconstructs for each goal g in G, the part of the subgoal graph relative to its corresponding goal g_C (step 1.2) in C. CS is the conflict set of g relative to Pl (step 1.1) and op_C is the operator of CS selected in C to achieve g_C (step 1.3). If op_C matches an operator op in the conflict set CS (step 1.4), op is applied to achieve g (step 1.4.1). The subgoals resulting from this application are matched against the subgoals of g_C relative to C (step 1.4.4) and the algorithm is called recursively (step 1.4.5). Finally, the algorithm checks if there are any delayed goals that can be solved with the augmented plan (step 1.4.6). If no operator in CS is matched by op_C, the achievement of the goal g is delayed (step 1.5.1).

The function **ReplayDelayedGoals** is shown in Figure 5.5. It follows the same steps as **ReplaySubgoalGraph** until 1.4.1. That is, the algorithm checks for every

```
ReplaySubgoalGraph(Pl, G, N, C)
1 for-each $g \in G$:
  1.1 $CS$ := conflictSet(g,Pl)
  1.2 $g_C$ := goalMatching($g$,$C$)
  1.3 $op_C$ := opDecisionAchieving($g_C$,$C$)
  1.4 if $\exists op \in CS$ with $matches(op, op_C)$
    1.4.1 $Pl$ := apply($op$,$g$,$Pl$)
    1.4.2 $subGP$ := subgoals($g$,$Pl$)
    1.4.3 $subGC$ := subgoals($g_C$,$C$)
    1.4.4 $SG$ := match($subGP$,$subGC$)
    1.4.5 $Pl$ := ReplaySubgoalGraph($Pl$,$SG$,$N$,$C$)
    1.4.6 $Pl$ := ReplayDelayedGoals($Pl$,$N$,$C$)
  1.5 else
    1.5.1 $N := N \bigcup \{g\}$
2 return $Pl$
```

Figure 5.4: The first phase of the replay process: reconstruction of the subgoal graph.

goal g, if the operator op_C achieving its corresponding goal g_C matches an operator in the conflict set of g (steps 1.1 - 1.3). If this is the situation, *op* is applied to achieve g (step 1.4.1). The reason for not performing steps 1.4.2 - 1.4.6 of **Replay-SubgoalGraph** is that applying *op* cannot generate any subgoals. That is, *op* is either a simple establishment or a planning operator (i.e., demotion, promotion, or separation). If *op* would have been an establishment with a new plan step, g would not have been delayed because performing an establishment of a new plan step only depends on the goal g. If *op* was not applicable relative to the new situation, *op* remains not applicable during the replay phase because new steps and constraints are added but none is removed.

```
ReplayDelayedGoals(Pl, N, C)
1 for-each $g \in N$:
  1.1 $CS$ := conflictSet(g,Pl)
  1.2 $g_C$ := goalMatching($g$,$C$)
  1.3 $op_C$ := opDecisionAchieving($g_C$,$C$)
  1.4 if $\exists op \in CS$ with $matches(op, op_C)$
    1.4.1 $Pl$ := apply($op$,$g$,$Pl$)
2 return $Pl$
```

Figure 5.5: Auxiliary function of **ReplaySubgoalGraph(P, G, N, C)**, the algorithm to reconstruct the subgoal graph.

The reconstruction of the subgoal graph (i.e., algorithm **ReplaySubgoalGraph**) is equivalent to standard replay; following the goal graph and applying each decision to achieve the goal is equivalent to following the derivational trace replaying the corresponding steps. For example, reconstructing the subgoal graph depicted in Figure 5.3 (b) results in the plan fragment depicted in the part (a) of the same figure. However, by using the subgoal graph, the basis is provided to reconstruct the justifications of the rejected decisions (see next section), to allow interactive planning and to perform intelligent backtracking during the completion phase of the adaptation process (see Section 5.8). The next section contains an example of reconstruction of the subgoal graph.

5.6 Reconstruction of the Justifications

Whereas the reconstruction of the subgoal graph corresponds to standard replay, the reconstruction of the justifications of the rejected decisions by CAPLAN/CBC during the adaptation phase is a novel characteristic in the *context of case-based, plan-space*

```
ReconstructJustifications($SG$, $SG_C$)
1 $D$ := reconstructedDec(SG)
2 for-each $d \in D$ with rejected($d_C$)
  2.1 $j_C$ := justifications($d_c$,$SG_C$)
  2.2 $dep_C$ := dependentDecisions($j_C$)
  2.3 if $dep_C \subset D_C$
     2.3.1 $j$ := newJustification($j_c$,$Pl$)
     2.3.2 assignJust($j$,$d$,$SG$)
     2.3.3 rejectDec($d$,$SG$)
3 return $SG$
```

Figure 5.6: Reconstruction of the justifications of the rejected decisions.

planning. The algorithm **ReconstructJustifications** performs the reconstruction process. **ReconstructJustifications** is performed after the subgoal graph, SG_C, of the case has been reconstructed relative to the new problem. As explained before, subgoal graphs represent plans (i.e., the partial-order plan obtained after following the assignments of the valid decisions). The input to the algorithm is the reconstructed subgoal graph relative to the new problem, SG, and the subgoal graph in the case, SG_C. All decisions (valid and rejected) in SG are collected in D (step 1). D_C denotes the set of decisions in SG_C that mapped D. That is, the decisions in SG_C that were reconstructed to obtain SG and d_C denotes the decision in D_C that maps d. As mentioned before, CAPLAN keeps track of the decision that added every assignment. Thus, given the justification j_C of a rejected decision d_C relative to SG_C, the set of decisions, dep_C, on which the assignments in j_C depends can be determined (step 2.2). If dep_C is contained in D_C, all assignments of j_C must have been reconstructed in SG. In this situation (step 2.3), the corresponding justification

j can be reconstructed relative to SG (step 2.3.1), j can be assigned as the justification of d (step 2.3.2) and d can be marked as rejected (step 2.3.3). In the last step of the algorithm, the updated subgoal graph SG containing the justifications of the rejected decisions is returned (step 3).

Example of Reconstruction of the Subgoal Graph and the Justifications. The reconstruction of the subgoal graph and the justifications can be illustrated by continuing with the example depicted in Figure 5.3. Recall that the reason for rejecting the demotion of s_2 to solve the threat $s_2 \overset{-}{\longleftrightarrow} (s_1 \rightarrow p@s_3)$ is that s_1 preceeds s_2 in the plan. Thus, a cycle occurs if s_2 is demoted because this operation orders s_2 before s_1. The fact that s_1 preceeds s_2 is consigned in the assignment a_1 in *decision-3*. As the algorithm **ReplaySubgoalGraph** reconstructs the subgoal graph by traversing it top-down, initially, either $p@s_3$ or $r@s$ are mapped to corresponding subgoals in the subgoal graph of the current situation. Suppose that $p@s_3$ is mapped to a subgoal $p'@s'_3$ and that *decision-1*, establishing $p'@s'_3$ with an step s'_1 is valid relative to the new situation. At this point, the subgoal corresponding to solve the threat $s_2 \overset{-}{\longleftrightarrow} (s_1 \rightarrow p@s_3)$ must be delayed. This threat is a subgoal resulting from making *decision-1* and as such is evaluated in the recursive call. This subgoal will be delayed because there is no operator solving it (the threat does not exists yet). Next, suppose that the goal $r@s_2$ is mapped against a goal $r'@s'_2$ and that *decision-2* establishing it with an step s_2 is valid. At this point, there are two possibilities: either, a negative effect of s_2 matches the precondition $p@s_3$ or not. In the former situation, the subgoal $s_2 \overset{-}{\longleftrightarrow} (s_1 \rightarrow p@s_3)$ matches $s'_2 \overset{-}{\longleftrightarrow} (s'_1 \rightarrow p'@s'_3)$. In the latter situation, this planning subgoal cannot be reconstructed (i.e., the threat does not occur in the new situation because of the binding constraints). Suppose that the planning subgoal can be reconstructed. In this situation the algorithm pursues to reconstructed the decision achieving it (this decision is not depicted in Figure 5.3). Eventually, the subgoal $q@s_2$ is checked. Suppose that it is mapped to a subgoal $q'@s'_2$ and that *decision-3* is valid. This decision corresponds to extablishing $q'@s'_2$ with the same step establishing $p'@s'_3$. That is, it is established with s'_1. Thus, s'_1 is ordered before s'_2 in the plan and, thus, *decision-4* (i.e., demoting s'_2) must be rejected in the new situation. Indirectly, this is what the algorithm **ReconstructJustifications** does at step 2.3; namely, checking that all elements in the justification are reconstructed relative to the new situation.

5.7 Validity of the Justification Reconstruction Claim

In the previous section we motivated with an example that a rejected decision in the case can also be marked as rejected relative to the current problem if all the assignments of its justifications can be reconstructed as well. In turn the assignments can be reconstructed if the decisions on which they depend can be reconstructed. We also saw that this checking is done by the algorithm **ReconstructJustifications**.

In this section we will take a closer look at the possible rejection decisions to explain why the justification reconstruction claim holds (see Claim 5.1). In general, there are three possible *failures* that can occur in SNLP (Kambhampati et al., 1996b):

Ordering failures. Ordering failures occur when there are two plan steps s_1 and s_2 such that $s_1 <_{\rightarrow} s_2$ and $s_2 <_{\rightarrow} s1$. That is, there is a cycle in the plan.

Binding failures. Binding failures occur when the binding constraints of the plan B are inconsistent. That is, when there is no possible instantiation of the variables with objects that is consistent with B. An example of such a failure occurs when the constraint *IsOfType(x,Table)* but no object in the problem is of type *Table*.

Establishment failures. This failure occurs when an open precondition $p@s$ can be established neither by an existing step in the plan nor by adding a new step in the plan.

If a taken decision d results in a failure, the decision is rejected and a justification is constructed. If an ordering failure occurs when a decision is taken, the decision corresponds to the application of a domain or a planning operator (excluding the separation operation). This means that a link of the form $s_1 \rightarrow p@s_n$ or $s_1 \rightarrow s_n$ is added to $\rightarrow$ and that currently in the plan there is a chain of the form $s_2 \rightarrow s_3$, $s_3 \rightarrow s_4$, ..., $s_{n-1} \rightarrow sn$ in $\rightarrow$.[2] Each $s_i \rightarrow s_{i+1}$ must be added by an assignment a_i that dependends on a decision d_i in the subgoal graph. Thus, the justification of the rejected decision is $\{a_1, ..., a_{n-1}\}$. Algorithm **ReconstructJustifications** says that the justification can be reconstructed relative to the new problem if *each decision d_i is reconstructed as a decision d_i' by the algorithm **ReplaySubgoalGraph***. In this situation, taken decision d will result in a failure relative to the new problem. A similar argument can be made to explain why the same holds for ordering and establishment failures.

5.8 Discussion of Complete Decision Replay

Figure 5.1 illustrates graphically, why a better performance is expected during the completion of the partial solution when complete decisions replay is performed compared to standard replay. A more technical explanation is based on the following fact: each time CAPLAN (and SNLP for that matter) makes a decision a consistency check has to be made to ensure that no failure will occur on the plan. If, as explained in the previous section, a new link is added to $\rightarrow$, CAPLAN needs to check that no cycle will occur. This check is typically much more expensive than the check that **ReconstructJustifications** as it frequently involves a constraint propagation

[2]The orderings can be caused by causal links, $s_i \rightarrow pi@s_{i+1}$, by protection links $s_i \rightarrow s_{i+1}$, or by combinations of causal and protection links as well.

process. The experiments will show that in fact the time effort to find the solution plan decreases when complete decision replay is performed compared to standard replay.

The next issue is related to the overhead caused by reconstructing the justifications. Clearly, if to find a solution plan, CAPLAN does not need to revise any of the decisions taken during replay, the justifications do not need to be reconstructed. In this situation, the overall problem solving process effort will be less if standard replay is performed instead of complete decision replay. Whether this happens or not depends on the domain, the size of the problems, and the retrieved cases. Our extensive experience with the domain of process planning and some artificial domains such as ART-1D-RES (see Appendix C) show that usually some decisions taken from the case need to be revised and that it does pay off to perform complete decision replay. One reason for this, is that until now similarity metrics in CBP have been static, so no learning takes place to improve them. In the next chapter we will present a framework based on feature weighting to dynamically improve the similarity metric. If a case is retrieved several times it is less likely that it will not fit into a solution of the new problem. On the other hand, until an adequate similarity metric is learned, it is likely that the case will not fit. In Chapter 7, a framework unifying complete decision replay and feature weighting will be presented, as a result of which CAPLAN is able to estimate whether the case will fit or not and dynamically decide to use complete decision replay or standard replay.

As discussed in Chapter 3, complete decision replay enables the user to interact with the system even after replay has taken place. This novel characteristic is based on the plan representation in CAPLAN which as we saw is based on the representation of plans in the subgoal graph and the construction of justifications. As these structures are reconstructed during replay, the functionality of CAPLAN is naturally inherited in CAPLAN/CBC. In particular the user may

- Reject decisions. That is, the user may declare the decision made to achieve the goal should be rejected. By using the dependencies represented in the subgoal graph, the rejection is propagated so that the parts of the plan that are not affected remain without having to plan from the scratch.

- Select goals and make decisions. The user is able to select which goal to achieve next or/and to make a decision from the conflict set.

The subgoal graph enables the base-level planner to use more powerful backtracking mechanisms than chronological backtracking (Weberskirch, 1995). These mechanisms can be used to complete the partial solution obtained after replay as a result of the reconstruction of the subgoal graph made at the first phase of complete decision replay.

Another aspect that has not been discussed is multi-case replay. In Chapter 8 a study of kinds of merging for multiple cases in the context of CBP is presented.

Chapter 6

Feature Weighting in Case-Based Planning

In Chapter 4, we presented dependency-driven retrieval, a method in which ordering restrictions of the new problem between the goals are used to perform an informed retrieval to preselect a set of cases. One or more cases are preselected for which the achieved goals match disjoint subsets of goals of the new problem and their dependencies are consistent with the ordering restrictions in the problem (see the ordering consistent condition, Definition 4.6). Thus, given an extended problem description $(I, G, \prec)$, dependency-driven retrieval preselects a set of cases based on the ordering restrictions and the goals of the problem $(G, \prec)$. In this chapter, we will see how CAPLAN/CBC makes a final selection between the preselected cases based on the initial features of the problem I.

Traditionally, initial features of the cases and the problem are compared based on static criteria. Some systems assign feature weights to rank the initial features based on an analysis of the particular solution found. Others classify the initial features between relevant and non relevant depending on whether the features contributed to the solution plan of the cases or not (see Section 2.6). Common to all of them is that these criteria are stated when the cases are created and is static. By static we mean that the similarity metric between a case and a problem is the same independent of previous retrieval episodes.

In many domains, initial features of the cases are key for the success of the adaptation process; if these features are matched, in the current problem the adaptation is likely to be a success. On the other hand, if these features are not mached the adaptation will result in a failure.[1] Moreover, features can be ranked according to their relative importance. A major difficulty is that obtaining this ranking depends on several factors making its apriori determination not feasible. For a given case, these factors are:

[1]Later in this chapter the concepts of "success" or adequateness and "failure" will be carefully defined. As for now they can be thought as a measure of the adaptation effort.

- the problem description,
- the particular solution plan found,
- the adaptation method, and
- the domain theory.

The first two factors were considered in previous work to determine the relevance of the features (Veloso and Carbonell, 1993) and to estimate a rank between them (Kambhampati, 1994). The other two factors are also important and must be taken into accout. For instance, the absence of an initial feature can be overcomed with little adaptation effort whereas the absence of another feature may be isurmountable for the adaptation method making useless the guidance provided by the retrieved case.

In CAPLAN/CBC we developed a case-based planning framework based on considering the feedback from the adaptation method to tune the rank of the relevant features (Muñoz-Avila and Hüllen, 1996; Muñoz-Avila et al., 1997). The rank of a feature is given by its weight, a non-negative number reflecting the relative importance of the feature in the case. Tuning the rank of a feature means in this context updating its associated weight. Deciding whether to retrieve a case or not is based on the current distribution of the feature weights. This distribution can be seen as a hypothesis about the best distribution of the feature weights. This hypothesis is tested in subsequent problem solving episodes. Depending on the outcome of the retrieval the hypothesis is reinforced or punished. Reinforcement and punishment are done by updating the feature weights and, thus, changing their distribution in the case.

6.1 The Weighted Similarity Metric

Base of the framework to update the feature weights is the weighted similarity metric. This metric counts the feature weights of the common features of the case and the problem. Each feature i has an associated weight $\omega_{i,C}$ depending on the particular case C. $\omega_{i,C}$ is a non-negative value reflecting the relative importance of the feature to the case. In particular, the weight of all non-relevant features can be seen as to be zero.[2]

Definition 6.1 (Weighted Similarity Metric) *The weighted similarity metric between a case C and a problem P, $sim^{wg}(C,P)$, is defined as:*

$$sim^{wg}(C,P) = \begin{cases} \sum_{i \in I \bigcap_\theta I_C} \omega_{i,C} & : \quad G_C\theta \subset G \\ 0 & : \quad otherwise \end{cases}$$

[2]Non-relevant features of a case are not taken into account. In fact, when a case is created, non-relevant features are even pruned from the problem description (see Section 2.6).

where $I \bigcap_\theta I_C$ denotes the set of all features in I_C matching a feature in I with a substitution θ.

The weighted similarity metric counts the weights of the features in the case matching a feature in the current problem. When a case is created, the weight of the relevant features is assigned. This initial assignment is updated in subsequent problem solving episodes in which the case is retrieved. It is possible that two different cases match the same subset of features of the problem. Their similarity, however, may be very different depending on the corresponding weights. The meaning of the weighted similarity metric can be explained with the following statement:

If the weights of the features of a case C were normalized so that $\omega_{i_1,C} + \omega_{i_2,C} + ... + \omega_{i_n,C} = 1$, then the factor $\sum_{j \neq k} \omega_{j,C} / \sum_i \omega_{i,C}$ expresses the *reliability* of making an adequate retrieval, when the feature i_k is the only one not matched in the new problem.

As mentioned in Section 3.2.2, searching for the most similar case has a prohibitive time cost. Instead, based on the weighted similarity metric, a retrieval condition is stated to determine if a case is to be retrieved (this condition is an extension of the one in PRODIGY/ANALOGY by considering feature weights):

Definition 6.2 (Weighted Retrieval Condition) *Given a problem P, a case C meets the weighted retrieval condition, $SIM^{wg}(C,P)$, if and only if $G_C\theta \subset G$ and*

$$(sim^{wg}(C,P)/sim^{wg}(C,C)) \geq thr_{ret}$$

where thr_{ret} is a predefined threshold, called the retrieval threshold.

This means that if the weighted proportion of features in the case matching features in the current problem is greater than the retrieval threshold, the case is retrieved.

Weighting Model. For updating the feature weights, a feedback model based on incremental optimizers is used (Salzberg, 1991; Wettschereck and Aha, 1995). Each case, C, contains two counters: k^C and f^C. The first one indicates the number of times a case was adequately retrieved, and the second one the number of times in which not. The weight, $\omega_{i,C}$, of a feature i is updated according to the following equations,

$$\omega_{i,C} = \begin{cases} \omega_{i,C} + \triangle_{k^C,f^C} & : \textit{failed retrieval} \\ \omega_{i,C} - \triangle_{k^C,f^C} & : \textit{adequate retrieval} \end{cases}$$

where $0 \leq \triangle_{k^C,f^C} \leq \beta \times n^C$. The number of features in the initial state of C is denoted by n^C. Thus, the change in the weight of the features is bound by a factor,

$\beta \times n^C$, directly proportional to the number of features in the case ($\beta \geq 1$). If the value of ω_i^C is smaller than 1, then $\omega_{i,C}$ is assigned the value 1 and the weights of the other features in the case are incremented proportionally. The formula to compute the incremental factor $\triangle_{k^C,f^C}$ is as follows:

$$\triangle_{k^C,f^C} = \begin{cases} \beta \times n^C - (k^C/f^C) & : \quad k^C < f^C \\ \beta & : \quad k^C = f^C \\ \beta \times f^C/(k^C) & : \quad k^C > f^C \end{cases}$$

The incremental factor $\triangle_{k^C,f^C}$ depends on the values of k^C and f^C in the following way: the larger the ratio of k^C to f^C, the smaller is the value of $\triangle_{k^C,f^C}$. Thus, as the number of adequate retrieval episodes increases, the effect of a retrieval episode on the feature weights decreases. In contrast, the smaller the ratio of k^C to f^C, the closer is $\triangle_{k^C,f^C}$ to $\beta \times n^C$. Thus, the effect is the opposite: the larger the ratio of f^C to k^C, the higher the value of $\triangle_{k^C,f^C}$ (i.e., $\triangle_{k^C,f^C}$ comes closer to $\beta \times n^C$).

6.2 Analysis of Case-Based Planning Episodes

In the previous section we presented the weighted retrieval condition and the feature weighting model used to update the feature weights. Two issues will be explored in this section:

- Stating if the retrieval of the cases is adequate or a failure.
- Identifying the features whose weights must be updated.

Both issues are used by the weighting model. The first issue determines if the feature weights are to be increased or decreased. The second one indicates for which features the weight must be updated.

As described in Section 3.2.4, the notion of retrieval failure is based on the definition presented in (Ihrig and Kambhampati, 1996a), which we now recall:

Definition 6.3 (Retrieval Failure, Adequate Retrieval, Skeletal Plan) *Given a solution plan Sol of a problem P obtained by adapting a case C with standard replay, then the retrieval of C is a failure with respect to P and Sol if at least one decision replayed from C was revised by the first-principles planner to obtain Sol. Otherwise the retrieval of C is said to be adequate. The partial solution obtained after replay is called the skeletal plan.*

This definition says that the retrieval of a case is considered adequate if the subplan obtained after replay can be extended to a solution plan of the current problem. In CAPLAN/CBC we distinguish between an adequate and a beneficial retrieval. The latter measures the adaptation effort and determines whether a new solution is to be stored as a new case. We will examine this issue in Chapter 7.

Example of a Retrieval Failure. Before continuing, we recall the example presented in section 3.5, in which a complete problem solving episode is illustrated. In this problem episode a problem is given consisting of five goals. Three of which are matched by the retrieved case $C5$ (namely, *processed(H)*, *processedH1(U_1)* and *processedH2(U_1)*). The other two remain unsolved after replay (i.e., *processedH1(U_2)* and *processedH2(U_2)*). In addition a feature in the case does not matched any feature in the problem; namely, the feature *available(rrt)* indicating that a tool of type "right cutting tool" is available. The skeletal plan obtained after replay is depicted in Figure 6.1. The precondition regarding the use of a tool of type "right cutting tool" is left open (this situation is depicted by a question mark, "?", adjacent to step 6). The retrieval of the case results in a failure because the open precondition cannot be achieved with additional planning (there are no operators producing a cutting tool). Thus, step 6 needs to be revised.

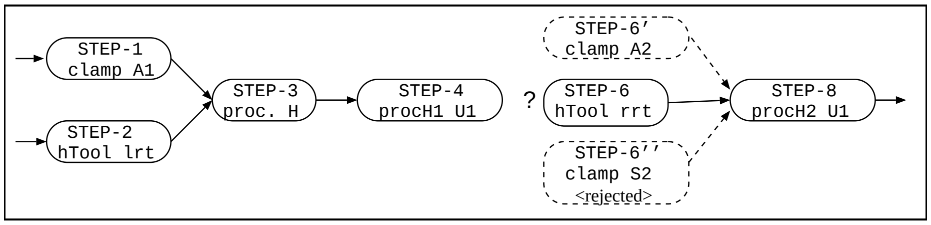


Figure 6.1: Partial solution obtained after replay (i.e., an skeletal plan).

Non-matched Features in the Skeletal Plan. CAPlan/CbC identifies the relevant features whose weights must be recomputed by examining the contribution of the feature to the solution of the case by taking as basis the skeletal plan. For example, the precondition *available(rrt)@STEP-6* of the skeletal plan shown in Figure 6.1 was not achieved during replay because it cannot be established with *start*. Thus, as the retrieval failed, the weight of the corresponding feature in the case, *available*(*rrt*), is increased by the factor $\triangle_{k^{C5},f^{C5}}$.

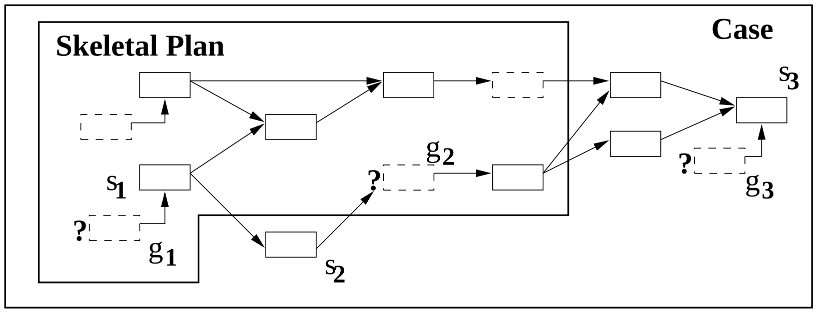


Figure 6.2: Abstract configuration of a skeletal plan and a case

Non-matched Features outside the Skeletal Plan. In more complex situations non-matched, relevant features may not be included in the skeletal plan. Consider, for example, the feature g_3 in the abstract situation described in Figure 6.2.[3] A question arises, namely, whether the weight of *g3* must be recomputed or not. A first possibility is to update the weights of all non-matched features. However, this does not takes into account the particular adaptation episode; as replay starts from the goals reconstructing the subgoal graph relative to the current problem, the fact that some features were non-matched may have no influence whatsoever on the resulting skeletal plan. This is illustrated in Figure 6.3. The decision d_1, which reduced the subgoal sg, was not reconstructed in the current situation because $d1$ is not applicable in the new problem. That is, there is no binding of the variables in *op* which is consistent with the binding constraints of the plan and the constraints of *op*. Suppose, in addition, that the subgoal sg_1 was reduced with a decision corresponding to applying a simple establishment with *start*. That is, sg_1 is achieved directly with a feature i in the initial state of the problem. If in the case, the binding of the variables in *op* does not involve any of the arguments of i, i has no influence on the fact that d_1 could not be reconstructed in the new situation. Further, if this was the only place where i was used and i is one of the unmatched features, then the weight of i should not be updated.

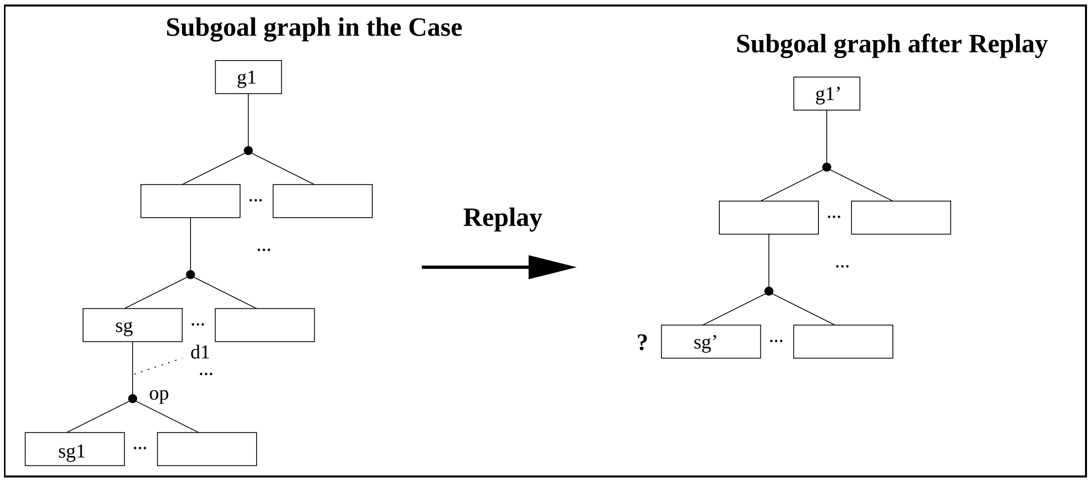


Figure 6.3: Abstract example of a replay episode.

Filtering Non-matched Features. CAPLAN/CBC performs a careful analysis based on the resulting skeletal plan and the original case. This analysis is called *filtering* as a subset of the non-matched features is selected. Filtering is performed

[3]Continuous boxes represent plan steps and continuous lines represent the order of execution among the plan steps. Dashed boxes represent preconditions of steps. Preconditions remaining unsolved after replay are labelled with a question mark, *?*.

by the function $FilterFeatures(C, Sk)$ that returns a set, *Expl*, of non-matched, relevant features whose weight must be recomputed. The set *Expl* meets the following condition:

***Expl* explains the unsolved preconditions in the partial plan.**

Explaining Open Preconditions. In the context of retrieval, explaining an open precondition p, means finding the features in the case such that if they would have been also present in the new problem, the precondition p would have been achieved in the skeletal plan. The explanation for an unsolved precondition that is established with the initial state in the case is the feature used to achieve the precondition. Examples of such preconditions are *available(rrt)@STEP-6* that occurs in the skeletal plan shown in Figure 6.1, and g_1 that occurs in the skeletal plan shown in Figure 6.2. If p is established with a plan step in the case different from *start* (i.e., p is not established with an initial feature of the problem), a careful analysis must be done to explain it. For example, in Figure 6.2, it is assumed that g_2 remains unsolved because a failed establishment with the plan step s_2 occurs.

The function $FilterFeatures(C, Sk)$ is shown in Figure 6.4. It receives the retrieved case, C, and the skeletal plan, Sk, and returns an explanation, *Expl*. This function uses two global variables: Arg and P. Arg contains all arguments of features in *Expl* and is computed dynamically. Initially, Arg and *Expl* are empty (steps 1 and 2). The idea of the function $FilterFeatures(C, Sk)$ is to examine each open precondition in the skeletal plan to obtain an explanation. To accomplish this, P contains the open preconditions in Sk that have not been examined yet. Initially, P is assigned all unsolved preconditions in Sk.

```
FilterFeatures($C, Sk$)
1. $Expl \leftarrow \{\}$
2. $Arg \leftarrow \{\}$
3. $P \leftarrow unsolvedPrecond(Sk, C)$.
4. while (exists $p \in P$ with $isEstabI(p, C)$)
    4.1 processEstab(p)
    4.2 P $\leftarrow$ P $-$ $\{p\}$
5. while ($P \neq \{\}$)
    5.1 processInconsistency(P,C,Sk)
    5.2 P $\leftarrow$ P $-$ $\{p\}$
6. return Expl
```

```
processEstab(p)
1'. Expl $\leftarrow$ Expl $\bigcup \{p\}$
2'. $Arg \leftarrow Arg \bigcup argIn(p)$
```

```
processInconsistency(p,C,Sk)
1''. $I \leftarrow argsInconsistStep(p, C)$
2''. If $I \subset Arg$ then skip
3''. while ($I - Arg$) isNotEmpty
    3.1'' Let $c_i \in I - Arg$
    3.2'' p' $\leftarrow searchFeat(p, c_i, C)$
    3.3'' processEstab(p')
```

Figure 6.4: Algorithm for filtering features.

Recall that a precondition, p, can be achieved by establishing it either with the initial state (i.e., with the dummy step *start*) or with a plan step associated with an operator (see Definition 2.5). Thus, there are only two possible reasons why p was achieved in the case and not in the skeletal plan:

1. The establishment with *start* cannot be performed because there is no feature in the initial state of the new problem that matches p. Examples of failed establishment with *start* are $available(rrt)$*@STEP-6* in Figure 6.1 and g_1 in Figure 6.2. This failure is considered in the first control block of the algorithm (step 4): each precondition, p, whose plan step in C is established with the initial state (i.e., $isEstabI(p, C)$ is true), is stored in *Expl* (step 1') and its arguments are collected in Arg (step 2'). The idea is that if the feature would have been present in the initial state, the precondition p would have been achieved by establishing it with *start*.

2. The establishment with a plan step can not be performed because otherwise a failure occurs (i.e., an inconsistency is introduced). This means that the step used in the case to establish the precondition p cannot be used because the effects of the corresponding operator does not match p modulo the bindings constraints B in the current plan and the constraints of the operator. Thus, if the operator is applied an inconsistency in the bindings occurs. This means, for example, that there are two variables x_1 and x_2 bound with the constants a and b respectively, and the constraint $x_1 = x_2$ is stated in the plan step. For example, in Figure 6.2, it is assumed that s_2 can not be replayed because a failed establishment with a plan step occured. Thus, the precondition g_2 remains unsolved in the skeletal plan. This failure is considered in the second control block of the algorithm (step 5). If the inconsistent arguments of the plan step are included already in Arg, nothing is done (step 2"). Otherwise, two steps are performed: first, each argument, c_i, that was not included in Arg already is added to Arg (steps 3.1", 2'). Second, for each argument c_i, the feature found by calling the function $searchFeat(p, c_i, C)$ is added to *Expl* (steps 3.2", 1'). This function returns a feature achieving a precondition, p', in the case that has c_i as argument and such that the distance[4] between p' and p is minimal. The point here is that if the feature binding the inconsistent variables would have been present in the initial state, the inconsistency would not have occured and the plan step would have been applicable. Thus, the precondition p would have been established with the plan step.

In summary, *one or more features occuring in the case but not in the new problem are identified as the explanation for not solving the precondition* p *in the skeletal plan.* In the case C, p is established either with *start* or with a plan step associated with an operator. In the first situation, the explanation is constituted by the feature in C matching p. In the second situation, the explanation is constituted by the features binding the variables, which caused the inconsistency that made the plan step inapplicable in the new problem.

[4]The number of arcs of the shortest path connecting two preconditions in the plan.

Updating the Weights of Features in *Expl*. Each of the features in *Expl* is updated according to the outcome of the retrieval as shown in the algorithm of Figure 6.5. C and Sk denote the case and the skeletal plan. If the skeletal plan is extendable to a complete plan (step 2), the complete plan is a solution plan of the problem because all preconditions are achieved; in particular the preconditions of *FINISH*, which are the goals of the problem (see Section 2.3). In this situation, the weight of each feature in *Expl* is incremented by a factor $\omega_{i,C}$. Otherwise, the weight of each feature in *Expl* is decremented by the same factor.

```
evaluateAdaptation($C, Sk$)
1. Expl ← FilterFeatures(C,Sk)
2. If (isExtendible(Sk))
   For-each $i \in Expl$
      $\omega_{i,C} = \omega_{i,C} + \triangle_{k^C, f^C}$
3. Else
   For-each $i \in Expl$
      $\omega_{i,C} = \omega_{i,C} - \triangle_{k^C, f^C}$
```

Figure 6.5: Algorithm evaluating the adaptation effort.

6.3 Feature Context and Trivial Serializability

As mentioned in the beginning of this chapter, several factors affect the ranking of the features in a case according to its importance. These factors, namely, the problem description, the particular solution plan, the adaptation method and the domain theory together with the description of the current problem form the *context of the case features*. The adaptation method used in CAPLAN/CBC is either standard or complete decision replay. Thus, the adaptation method is an invariant part of the context independent of the other factors because the notion of retrieval failure is the same for both forms of replay. In this section, we will examine the role of the domain theory in determining the feature context. We will identify situations in which updating the feature weights is done although the outcome of the retrieval is determined by other factors in the context distinct from the initial features. We will provide a characterization to identify these situations based on the domain theory. This characterization will be illustrated with two examples. In the first one updating the feature weights is adequate but in the second one it is not.

A Case in the Logistics Transportation Domain. Consider the initial situation in the logistics transportation domain (Veloso, 1994) illustrated in Figure 6.6 (a). In this situation there are three post offices A, B and C. In post office A there is a package p_1 and a truck. In post office B there is a package p_2. In the final situation

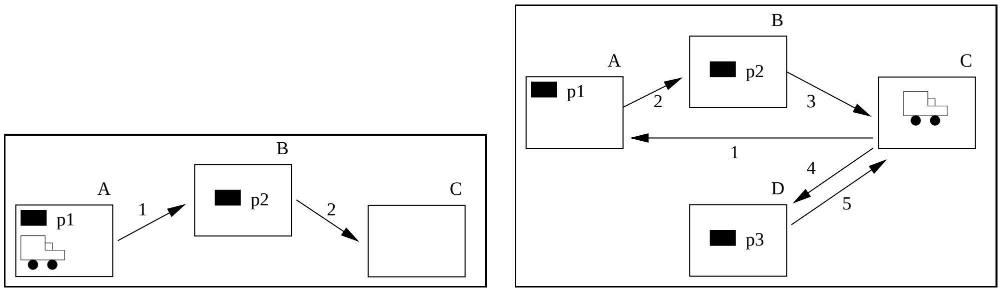


Figure 6.6: Initial situation of (a) the case and (b) the new problem.

both packages, p_1 and p_2, must be located at C. There are basic restrictions that any solution must meet and as such form part of the domain theory: (1) only trucks can move between post offices, (2) to load a package in a truck, both have to be located at the same post office, and (3) to unload a package from a truck in a certain office, the truck must be located at that post office. A possible solution is to load package p_1 at A, move the truck from A to B, load package p_2 in the truck, move the truck from B to C and unload both packages (the arcs show the path followed by the truck, the numbers indicate the order). Suppose that this problem and solution are stored as a case.

Example of a Situation in which Updating Feature Weights is Adequate. Consider a new problem with the initial situation illustrated in Figure 6.6 (b). In the final situation the three packages must be located at C. If the case is used to solve the new problem, the truck follows the path illustrated by the arcs, collecting at each post office the corresponding package, leaving the packages p_1 and p_2 in C as indicated in the case. Finally, package p_3 is loaded and moved to C. In this situation, the retrieval of the case is *adequate* because steps taken from the case (2 and 3 in the new problem) can be extended to solve the new problem. The problem solved in the case was not totally contained in the new problem: in the case, the truck is located in the same post office as a package whereas in the new problem, the truck is located in a post office with no packages. Technically, this means that some initial features were unmatched by the initial features of the new problem. If we take the unmatched and matched features of the case as input for a weighting model, the weight of the unmatched features found by the algorithm *FilteringFeaures* is decreased relative to the weight of the other features in the case because their absence did not affect the reusability of the case.

Example of a Situation in which Updating Feature Weights is not Adequate. Now consider the same case and problem as before, but suppose that additional restrictions have been added to the domain theory: (4) trucks must not be moved into the same post office more than once and (5) problem-specific restrictions such as not allowing the truck to move from D to A in Figure 6.6 (b). These restrictions are made to improve the quality of the plan. Clearly, the path illustrated in Figure 6.6 (b) violates restriction (4) because the truck is moved to post office C twice (arcs 3 and 5). This means that the solution of the case must be revised. In particular, moving the truck from B to C is revised and instead it must be moved from B to D, where package p_3 is loaded. Finally, the truck is moved from D to C, where the three packages are unloaded. In this situation, the retrieval of the case is considered to be a *failure* and the weight of the unmatched features is increased relative to the weight of the other features in the case. However, this does not reflect the real reasons for the failure: even if the truck is located at A, the plan must still be revised. The real reason is that in solving the additional goal, to locate p_3 in C, a conflict with the solution of the case occurs. That is, *the goal interacts negatively* with the case. This means that *there are factors that affect the effectiveness of reusing cases different than the initial features.* As a result, the strategy of updating the weights of the features based solely on the matched and unmatched features of the case becomes questionable.

Characterization of the Situations. The difference between the standard specification of the logistics transportation domain and the extended (i.e., with restrictions 3 and 4) is that goals in the first one are trivially serializable but in the second one not necessarily. This motivates the following claim:

> **Claim 6.1 (Context-Simplified)** *In domains where goals are trivially serializable, the feature context are the initial features of the problem and of the case, the goals common to the problem and the case, and the solution of the case.*

This claim essentially says that the additional goals do not affect the reliability of the retrieval. As a result, weighting models on initial features can be used. To show this, let G^C, G denote the goals of the case and of the problem respectively, then the subplan achieving $G^C \cap G$ in the case is taken and extended relative to the initial situation of the new problem (in the example, this extension corresponds to moving the truck from C to A, i.e., arc 1). Once the plan achieving $G^C \cap G$ has been generated, it can be extended to a plan achieving G because the goals are trivially serializable (i.e., arcs 4 and 5). Of course, retrieval failures will still occur if the subplan achieving $G^C \cap G$ in the case cannot be extended to solve these goals relative to the initial situation of the new problem. But the point is that such a failure is due to the initial features and not to the additional goals.

Table 6.1 summarizes this result (Sol^C represents the solution plan of C). If goals are trivially serializable, only goals that are common to the problem and the case need to be considered. However, if goals are not trivially serializable, additional goals in the problem and the case might affect the reusability of the case. This result establishes a direct relation between the concept of trivial serializability and feature

Goals in Domain are	**Context**
trivially serializable	$Sol^C + I^C + I + (G^C \cap G)$
not trivially serialiable	$Sol^C + I^C + I + G^C + G$

Table 6.1: Context according to the characteristics of the goal interactions in the domain.

context and shows the feasibility o feature weighting in case-based planning. Notice that the claim is independent of the particular kind of planner being used; all it requires are the goals to be trivially serializable for that particular kind of planner and reuse based on replay.

Until now, no mention has been made about whether the fact that CAPLAN/CBC uses extended problem descriptions plays any role on determining the context. Based on the ordering restrictions $\prec$ of the problem, a weakened form of trivial serializability will be defined in the next chapter. This form of trivial serializability will be used to weaken the conditions regarding trivial serializability of Claim 6.1. We closed this section by defining goals interacting negatively with a case.

Definition 6.4 (Goal Interacting Negatively) *Given a problem (I, G) and a case such that $G^C \subset G$, where G^C are the goals achieved in C, a goal, g, in $G - G^C$ interacts negatively with C if for any skeletal plan, $Skel$, obtained after replaying C relative to (I, G), $Skel$ cannot be extended to a solution plan achieving $G^C \bigcup \{g\}$.*

6.4 Handling Arbitrary Planning Domains

There are several domains for which goals are known to be trivially serializable (Barrett and Weld, 1994; Kambhampati et al., 1996a). However, not for every domain can be supposed that its goals are trivially serializable. A typical example is the logistics transportation domain if restrictions (4) and (5) are considered. Notice that even with these restrictions, there are several situations in which the retrieval is adequate. For example, suppose that we have the same case as before and that the initial situation of the problem corresponds to a slight modification of the situation given in Figure 6.6 (b): p_3 is located at C and in the final situation, p_1, p_2 must be located at C and p_3 at D. In this situation, the retrieval of the case is adequate because the subplan achieving the first two goals can be extended to a plan achieving these two goals relative to the new problem by moving the truck from C to A (i.e., arc *1*). In addition, this subplan can be extended by loading p_3 in the truck, moving the truck from C to D and unloading the package (i.e., arc *4*). In this situation, the weights of the features can be updated because the additional goals do not interact negatively with the case. We will now show how explanation-based learning methods can be used to detect dynamically the situations in which the failure was caused by goals interacting negatively with the cases.

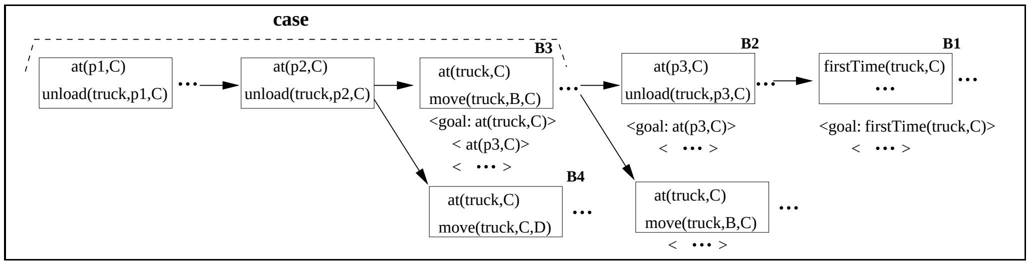


Figure 6.7: Example of the regression of explanations over a search tree.

Explanation-based Learning (EBL). Explanation-based learning (EBL) has been used to guide the search process in planning (Minton, 1988; Kambhampati et al., 1996b). The current state is viewed as a node in the so-called search tree.[5] If the node representing the current state has more than one successor node (i.e., there is more than one alternative to transform the current state), the planner has to chose one. If the choice was wrong (i.e., there is no node representing a solution in the subtree whose root is the chosen node), another neighbouring node has to be chosen. If all alternatives have been exhausted, the planner will have to go back to the predecessor of the node representing the current state and make another decision. This backtracking process is very expensive. For this reason, the search path can be analyzed to generate search control rules that explain the failure. When the same situation is encountered the planner will avoid making the choice known to be wrong.

Detecting Negative Interacting Goals with EBL. The basic idea is to use EBL to detect situation in which the goals contributed to the failure of the retrieval because they interact negatively with the case (in CAPLAN the EBL mechanism as presented in (Kambhampati et al., 1996b) was implemented (Roth-Berghofer, 1996)). Figure 6.7 sketches the search tree of the situation illustrated in Figure 6.6 if condition (4) is taken into account. The root of the tree is the left-most node. The search three grows from left to right. Nodes labelled with *Bn* indicate that backtracking has taken place. Nodes always show the goal being solved and the operator selected to achieve that node (e.g., in the root node the goal is *at*(p_1,*C*) and the operator *unload(truck,*p_1,*C*)). The nodes explored first are the ones taken from the case. As explained in the previous section, the subplan in the case achieving the first two goals can be extended relative to the initial situation of the new problem. However, the extended subplan cannot be further extended to achieve the third goal because of condition (4). From a technical point of view, a node will be reached that does not represent a solution and that either represents an inconsistent state or has no successors (not shown in Figure 6.7). At this node, EBL generates a so-called

[5]The meaning of the term "state" depends on the particular planning paradigm: for (Minton, 1988) "state" is a world state whereas for (Kambhampati et al., 1996b) it is a plan state (see Chapter 2).

initial explanation that describes the inconsistency or the fact that the node cannot be further transformed. *The initial explanations correspond to the justifications of the failed decisions in the sense of* CAPLAN (see Section 5.3). This initial explanation is *propagated* to the predecessor of the node. This propagation, called regression, results in an explanation of the failure in terms of the conditions that were valid at the predecessor node. If a node has more than one successor node its failure explanation is the conjunction of the regressed explanations of each successor node together with the goal being achieved. The obtained explanation can further be regressed when backtracking occurs using the same principle. In Figure 6.7, four of the backtracking nodes are shown and their corresponding regressed explanations are sketched between <>. Two observations can be made here:

1. To have a retrieval failure is equivalent to backtracking on nodes that were obtained through replay (Ihrig and Kambhampati, 1996a). For example the node *B3* replayed and backtracking on this node occurs. This means that the case could not be extended and, thus, a retrieval failure occurs.

2. *If backtracking is caused by goals interacting negatively with the case, these goals are present in the regressed explanation of the backtracking nodes.* This fact is illustrated in the regressed explanation $< goal : at(p_3, C) >$ of the node *B3*.

The second observation is very important because it clearly states how to identify if the failure is due to a negative interaction of a goal relative to the case; namely, if the goal appears in the regressed explanation at the corresponding node. We will illustrate why this statement is valid: notice that g will be included in the explanation of the backtracking at the node representing the state where g was pursued to be achieved (node *B2* in Figure 6.7). Because g is an original goal in the problem, there was no other node that added g. That is, every regression of an explanation containing g will contain g again. This shows why the explanation of *B3* includes the negatively interacting goal $at(p_3,C)$.

Evaluation Principle in General Domains. In domains that do not meet the conditions of Claim 6.1, the EBL method is used in the way described above to detect the situations where the goals interact negatively with the case. In these situations no feature weights are updated. In contrast, if the adequateness or failure does not involve any negatively interacting goal, the feature weights of the cases involved are updated. In other words, we are using EBL to determine the context (i.e., if the additional goals form part of it or not).

In Chapter 9 we will see that even in general domains feature weighting improves the reliability of the retrieval provided that a filtering is made with EBL to the feature weighting process as described in of this section.

6.5 Discussion of Feature Weighting

As will be shown through extensive experiments (see Chapter 9), feature weighting increases the reliability of the retrieval by reducing the number of retrieval failures. There are, however, two problems that may occur as a result of using feature weighting in case-based planning. The first one is the specialization of the cases, which can be defined as follows:

Definition 6.5 (Case Specialization) *A case C is specialized when there is a problem P such that:*

- *The weight of at least one feature in C is different than 1 and C does not meet the weighted retrieval condition relative to P.*
- *If the weights of all features in C are set to 1, C meets the weighted retrieval condition relative to P.*
- *If C is retrieved to solve P, the retrieval is adequate.*

The specialization of a case means that situations occur in which the case is not retrieved although it should. Depending on the previous problem episodes in which the case was retrieved, the distribution of the feature weights in the case may lead it to only recognize certain situations. As we will see in the experiments, specialization of cases seems an inevitable side effect of feature weighting. In the same experiments, however, we will see that in the average situation, the specialization of the cases tends to be low. Moreover, the increase in reliability is so significant that overall it pays off to perform feature weighting.

The second problem is that a case may need to be retrieved several times before retrieving it is reliable. This depends mainly on the problems which were given previously. Feature weighting -as any dynamic learning procedure- depends on the *order* in which the problems are given (also called *training examples* in this context); if a set of problems is given in a certain order, the case will learn a good distribution of feature weights rapidly whereas in other learning order a good distribution of feature weights may require several training examples. In CAPLAN/CBC this problem is handled by a synergistic integration of the retrieval and the adaptation procedure. This integration is called the *dual integration of retrieval and adaptation* and will be explained in Section 7.3. The principle is to use dependency-driven retrieval when non reliable cases are retrieved. The nonreliability of the case is compensated with the high degree of repair of this adaptation method. Complete decision replay is specially suited for situations in which replayed decisions need to be revised. That is, for situations in which retrieval failures occur. Moreover, when relialable cases are retrieved, the efficiency of the problem solving process is improved by performing standard replay as the partial solution obtained after replay is likely to be extensible to a solution plan.

Chapter 7

Integration Issues in CAPlan/CbC

In previous chapters we presented different aspects of CAPlan/CbC; first, we presented dependency-driven retrieval, which is the static retrieval technique developed to make a preselection of the cases based on the goals and ordering restrictions of the current problem $(G, \prec)$. Then, we observed the flaw of standard replay in plan-space planning; namely, that the degree of repair is low. Complete decision replay corrects this flaw by considering failed decisions made in the cases. If the justifications of a failed decision can be reconstructed, the decision must be also invalid in the current problem. Thus, this decision is not explored during the completion process. In the previous chapter, we presented a dynamic retrieval technique based on feature weighting. The features in the case are weighted according to their relevance to the solution in the case. The weighted similarity metric counts the weights of the features in the case matching features in the problem, I. The feature weights are recomputed according to the performance of the retrieved case in the case-based planning episode.

In this chapter, a cross examination of different issues involving the integration of dependency-driven retrieval, complete decision replay and feature weighting in CAPlan/CbC will be made. More concrete, the following issues will be discussed:

Organization of the Case Base. Retrieval in CAPlan/CbC is a twofold process. First, dependency-driven retrieval is performed to make a preselection of the cases. Second, feature weights of the preselected cases are considered to make a final selection. In the first section of this chapter, the organization of the case base will be presented, which enables CAPlan/CbC to perform the twofold retrieval process on an indexing structure instead of directly with the cases. The indexing structure extends the one presented in Section 4.6 by representing the feature weights.

Policy to add New Cases. The decision of whether to add a particular solution found as a new case depends on a measure of the benefit of the retrieval. In the second section of this chapter, this measure will be explained and contrasted with the notion of adequate or failed retrieval (see Definition 6.3).

Dual Integration of Retrieval and Adaptation. One of the very basic principles of CBR is that in the retrieval phase, one or more cases are selected which are adaptated to solve the current problem. In CAPLAN/CBC the retrieved cases are not only adapted into a new solution but they also determine which adaptation method to use, standard replay or complete decision replay.

The Domain Theory and Retrieval. Trivial serializability is a condition stated for problem descriptions (see Definition 2.9). In particular, it is based on a property that the goals in G must meet. Given that problems in CAPLAN/CBC are stated in the form of extended problem descriptions, the notion of trivial serializability can be restricted by considering the ordering restrictions $\prec$. In the fourth and last section of this chapter, the restricted form will be defined and discussed how the retrieval process can be adequated for domains in which $(G, \prec)$ meet the weakened condition.

7.1 Organization of the Case Base in CAPLAN/CBC

The case base in CAPLAN/CBC is a three level structure. The top level is formed by the type-representation table and the GDN (see Section 4.5). For each different type representation *typRep* of a set of goals achieved by a case, there is an entry in the type-representation table. Each entry points to a tree in the GDN in which all dependency classes of every set of goals achieved by a case and having the same type-representation are represented. Every path $[rt\ n_1\ ...\ n_m]$ in these trees such that rt is the root of T, n_1 is a son of rt, n_{i+1} is a son of n_i and n_m is a leaf of T meets the following condition: if G_i denotes the goals in n_i, then $[G_1,...,G_m]$ is the sequence of dependency classes for all cases indexed below n_m.

As explained in Section 4.6, the main advantage of the GDN is that the order consistency condition can be tested by traversing its trees; given a new problem, $(I, G, \prec)$, the GDN is used to identify a path $[rt\ n_1\ ...\ n_m]$ of a tree T such that all cases indexed below n_m meet the order consistency condition relative to $\prec$. These cases are the ones that are considered as preselected during the first phase of the twofold retrieval process.

We will now explain, how the indexing structure was conceived at the second level of the indexing structure, in which the feature weights are represented. Typically, initial features have been indexed by considering the frequence in which each feature occurs in the cases (Veloso, 1994); a tree structure, which we call feature-discrimination tree, is built reflecting the frequence with which the features occur. Features common to all cases are located in the root of the feature-discrimination tree. The successors of a node further discriminate the cases by containing features common to some cases. The initial features of any case indexed below a leaf n can be collected by following the path from the root to n. Figure 7.1 depictes a feature-discrimination tree for two cases, $C1$ and $C2$. The root contains the common features whereas the two leafs their differences. The motivation of this structure is to reduce the possibility of performing the same match between initial features more than once.

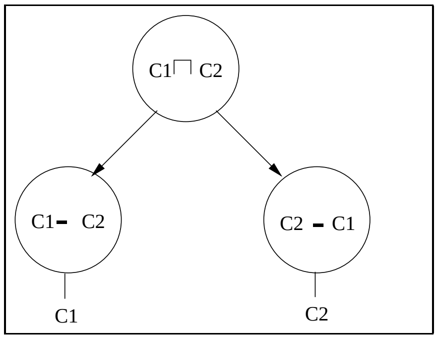


Figure 7.1: Feature-discrimination tree for two cases (Veloso, 1994).

Given the initial features of a new problem, the feature-discrimination tree is traversed starting from the root and counting for each case the percentage of its features that match features of the problem. If the percentage of features in a case matching features in the problem is greater than the retrieval threshold, the case is selected. If two or more cases meet this condition, one of them is alleatory selected and retrieved. This corresponds to the weighted retrieval condition but assuming that the weights of all features is 1 (see Definition 6.2).

As discussed in Section 6.1, each feature i in CAPLAN/CBC has an associated weight, $\omega_{i,C}$, that depends on the particular case C in which the feature occurs and reflects the importance of the feature in the case. In principle, the feature-discrimination trees could be used to search for a case meeting the weighted retrieval condition; instead of counting the number of features matched, their weights should be counted. However, by using the feature-discrimination trees, unnecessary matches may be performed as, frequently, the features that are most common have a small weight. Thus, matching them will not have a significant contribution in the process of finding a case meeting the weighted retrieval condition. For this reason, we conceived an indexing structure that considers the feature weights directly. The base of this structure is the notion of weight $\omega_{i,Coll}$ of a feature i relative to a collection of cases $Coll$.

Definition 7.1 *Given a collection of cases Coll, the weight $\omega_{i,Coll}$ of a feature i relative to Coll is defined by the formula:*

$$\omega_{i,Coll} = \sum_{C \in Coll} \omega_{i,C}$$

As we saw, each path from the root of a tree in the GDN to a leaf represents the sequence of dependency classes for all cases indexed below the path. The leaf of the path does not point directly to the cases, but to a structure that discriminates them based on their weighted features (see Figure 7.2). This structure consists of one or more intervals. Each interval contains at most a predefined number, *num*, of features occuring in the cases that are indexed below the leaf. To determine the intervals, the features i_{min} and i_{max} with the minimal and maximal weight relative to all cases in the collection are chosen. Two intervals are formed; in the first one all features with weight greater than $(\omega_{i_{min},Coll} + \omega_{i_{max},Coll})/2$ are grouped and in the second one all others. If more than *num* features are contained in an interval, the interval is partioned recursively in the same way. Thus, i_{min} and i_{max} are computed relative to the features in the interval. A feature-discrimination tree is used to index all features grouped in an interval. The number *num* is an input parameter to the system. It is typically set to 20 to maintain the size of the feature-discrimination trees relatively small. In the structure, the intervals are ordered starting from the one with the heavier weights and finishing with the one with the lighter weights. In Figure 7.2, w_1 is the weight of the heaviest features. An interval $(w_1, w_2]$ includes all weights between w_1 and w_2 but excluding w_2.

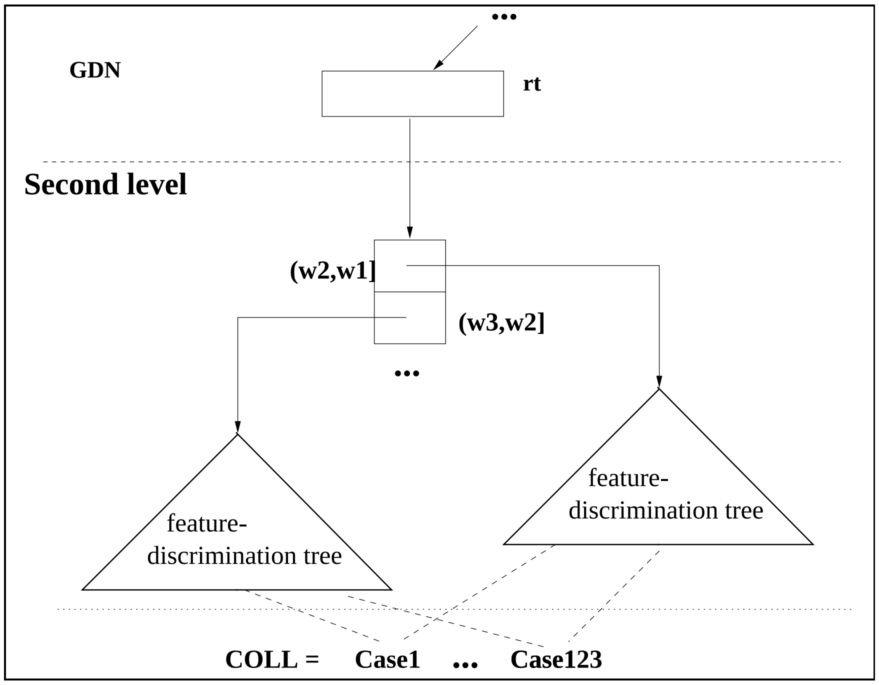


Figure 7.2: Fragment of the second level of the indexing structure in CAPLAN/CBC.

As we saw, during the first phase of the retrieval process, a path from the root to a leaf of a tree in the GDN is found. The collection of cases indexed below the leaf are known to meet the order consistency condition with a substitution θ. Thus, in the second phase, the structure pointed by the leaf is traversed to find a case in the collection meeting the weighted retrieval condition with the same substitution θ. To traverse this structure, CAPLAN/CBC keeps track of the weighted proportion of features in each case in the collection matching features of the current problem. The first interval visited is the one containing the features with the heavier weights. The process continues by visiting the next interval with heavier weights and so on. The process is finished when either a case is found meeting the weighted retrieval condition or the interval with the lighter weights has been visited and no case was found. In the latter situation, CAPLAN/CBC performs backtracking to find a new sequence of dependency classes and repeats the process again.

7.2 Policy to Create New Cases

As discussed in Section 3.2.4, the fact that a retrieval failure occurs does not necessarily means that the adaptation effort was large. Even if some decisions in the skeletal plan need to be revised, the adaptation effort may be small because of the high degree of repair of complete decision replay (see Chapter 5). More concrete, the more justifications of failed decisions can be reconstructed, the more decisions are marked as invalid in the conflict sets. Thus, if backtracking takes place, the number of decisions in a conflict set that are not invalid can be very small. As a result, the system may explore a relative small portion of the space to refit the skeletal plan into a solution plan. In this situation, clearly, there is no point in creating a new case with the solution of the current problem as this can be easily reconstructed with the available cases.

The opposite may occur as well; the retrieval may be considered to be adequate but still the adaptation effort may be quite large. This means that even though the skeletal plan was extended to a solution plan, the planning effort to find the solution plan was large. In this situation, the solution of the current problem found should be added as a new case in the case base. CAPLAN/CBC adds new cases only if the retrieval is considered nonbeneficial in the sense of the following definition:

Definition 7.2 (Beneficial Retrieval) *Given a plan Pl,* searchSpace(Pl) *counts the number of visited nodes when generating Pl.*

Given a solution plan Sol of a problem P obtained by adapting a case C, then the retrieval of C is beneficial with respect to Pl and C if:

$$searchSpace(Pl)/searchSpace(Skel) \leq thr_{ben}$$

where Skel *is the skeletal plan and* thr_{ben} *is a predefined threshold, called the benefit threshold.*

The value of the benefit threshold thr_{ben} is a parameter of the system. For example, if it is set to 2, the retrieval is considered nonbeneficial if the size of the search space explored to obtain *Sol* is at least twice as much the size of the one explored to generate *Skel*. To compute *searchSpace(Skel)* only the valid decisions reconstructed from the goal graph are counted. The reason for this is that the marking of decisions as failure is based on the reconstruction process of the justifications and formally does not involves the exploration of the search space (in the next section, the effort involved in reconstructing the justifications will be considered.) *searchSpace(Sol)* is the sum of *searchSpace(Skel)* and the size of the search space explored to refit *Skel* into a solution plan.

7.3 Dual Integration of Retrieval and Adaptation

As discussed in Section 5.8, complete decision replay reduces the completion effort of the skeletal plan by discarding decisions known from the retrieved cases to fail. Stating if a decision fails in the new problem knowing that it has failed in the retrieved case requires that its justifications can be reconstructed relative to the current problem. Reconstructing the justifications causes an overhead compared to standard replay. In complex domains, it pays off to perform complete decision replay as the retrieval fails frequently and, thus, decisions made to obtain the skeletal plan need to be revised. Still, if the outcome of the retrieval (i.e., whether it fails or it is adequate) can be predicted, the time for the adaptation process can be reduced. More concrete if the retrieval of a case is predicted as adequate, standard replay can be used instead of complete decision replay avoiding in this way the overhead caused by the justification reconstruction process.

The feature weights in a case indicate a hypothesis about their relative importance for the case. This hypothesis is tested in subsequent retrieval episodes, as result of which, the hypothesis is reinforcement or punishement of by updating the feature weights. If the hypothesis is correct, the retrieval is adequate. This means, that the decisions taken from the skeletal plan does not need to be revised and as such there is no point of performing complete decision replay. The problem is whether the outcome of the retrieval can be predicted or not. In CAPLAN/CBC related statistical information is maintained; namely, the number of adequate and failed retrieval episodes in which each case was involved. These numbers are used to determine the incremental factor $\triangle_{k^C,f^C}$, indicating to what extend the feature weights will be modified (see Section 6.1). The larger the proportion of adequate retrievals k^C relative to the failed retrievals f^C in a case C, the greater the possibility that the hypothesis about current distribution of the feature weights is correct and the less likely is that a retrieval failure occurs. Thus, this criterion can also be used to predict if standard or complete decision retrieval is to be used:

If the proportion of k^C to f^C is greater than thr_{rep}, standard replay is used. Otherwise, complete decision replay is used.

The number thr_{rep} is called the *replay threshold*, which is a parameter to the system. For example, if thr_{rep} is set to 4 to 1, standard replay is selected to adapt the retrieved case if from every five retrieval episodes involving the case four were adequate. Still, complete decision replay is key for the efficiency of the case-based, problem solving process. There are two reasons for this:

- Several retrieval episodes usually take place before a good distribution of the feature weights are learned.
- In complex domains, new cases are frequently learned.

In resume, the retrieval phase plays a dual role in CAPLAN/CBC; namely, the usual role in CBR of selecting one or more cases from the case base and a new role of determining which adaptation method should be used.

7.4 The Domain Theory and Retrieval

The notion of trivial serializability is based on the traditional specification of problem descriptions. That is, as a pair (I, G). However, if problems are specified by extended problem descriptions $(I, G, \prec)$, this notion can be simplified. By definition, if $\prec$ is valid, any solution plan for this problem meets the ordering restrictions in $\prec$ (see Section 4.1). Thus, instead of considering all permutations of goals as in trivial serializability, only permutations of goals that extend $\prec$ can be considered.

Definition 7.3 ($\prec$-Consistent Permutation) *A permutation π on a partially ordered set of goals, $((g_1, g_2, ..., g_n), \prec)$, is $\prec$-consistent, if, for any two goals, g_i, g_j, such that $g_i \prec g_j$, then, $\pi[g_i]$ is ordered before $\pi[g_j]$.*

Accordingly, we will restrict the definition of trivial serializability to consider only permutations that are $\prec$-consistent. In this way, we exclude from consideration permutation orders to achieve the goals that are not to be followed by the planner.

Definition 7.4 ($\prec$-Constrained Trivial Serializability) *A partially ordered set of goals, $((g_1, g_2, ..., g_n), \prec)$, is $\prec$-constrained trivially serializable, if any $\prec$-consistent permutation on these goals is a serialization order (modulo a class of plans $\mathbf{P}$).*

Notice that, *for deciding if a set of goals is $\prec$-constrained trivial serializable, the number of permutations required to be serially extensible is less than the number of permutations required for deciding if they are trivial serializable.* The reduction in the number of permutations to be considered is directly proportional to the degree of ordering defined by $\prec$. In the limit, if the set of goals is totally ordered, only one permutation, the one defined by $\prec$, must be inspected to decide if they meet this property. Thus, the following proposition can be stated:

Proposition 7.1 *Let $((g_1, g_2, ..., g_n), \prec)$ be a partially ordered set of goals. If g_1, g_2,...,g_n is trivially serializable, then $((g_1, g_2, ..., g_n), \prec)$ is $\prec$-constrained trivially serializable.*

Example. As mentioned before, in the domain of process planning, the ordering restrictions $\prec$ are stated by the geometrical reasoner. In (Muñoz-Avila and Weberskirch, 1996c) it is formally shown that goals corresponding to processing the areas of a workpiece are not trivially serializable modulo the class of elastic protected plans to process workpieces. However, in the same work it is shown that the goals are $\prec$-constrained trivially serializable modulo the class of elastic protected plans to process workpieces. An intuitive explanation can be given by recalling the example presented in Section 3.5.1. In that example the workpiece depicted in Figure 7.3 is to be processed. In particular, the horizontal outline H and the undercut U_1 must be processed. In addition, it is known that the horizontal outline H is to be processed before the undercut U_1, that is, *processed(H)* $\prec$ *processedH1*(U_1) holds. The reason for this is that H covers U_1. Thus, U_1 cannot be accessed until H is removed. If U_1 is processed before H, the precondition to process U_1 requiring the tool holder machine to be free will be established with *start* (i.e., is an initial feature of the problem). This establishment needs to be revised when ordering the processing step of H before the one of U_1. The precondition to process U_1 requiring the tool holder machine to be free must be established with an step unmounting the tool, which was mounted to process H.

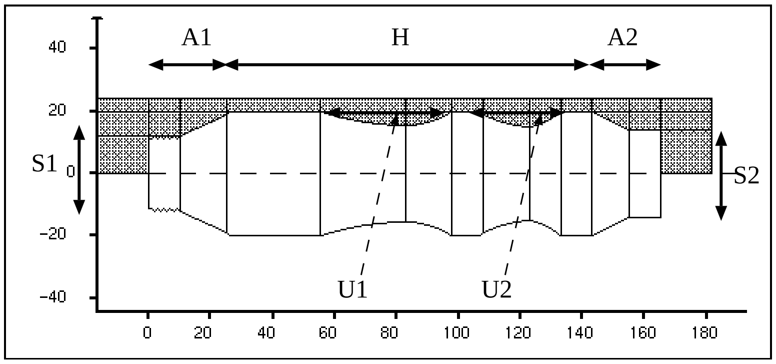


Figure 7.3: Display of a rotational symmetrical workpiece.

7.4.1 Dependency-Driven Retrieval and $\prec$-Constrained Trivial Serializability

In this section, we will discuss how the order consistency condition can be modified to consider domains that are known to have goals being $\prec$-constrained trivially serializable but not trivially serializable.

Example Motivating the Modification of the Retrieval Condition. Consider a problem consisting of three goals, g_1, g_2, g_3. Suppose that any solution must meet the ordering restriction $g_1 \prec g_3$ and that g_1, g_2, g_3 are known to be $\prec$-constrained trivially serializable but not trivially serializable. If a case C achieves the goals g'_1 and g'_3 such that g'_1 and g'_3 match g_1 and g_3 and $g'_3 <_C g'_1$ does not hold, C is still an adequate candidate. The reason for this, is that during completion new ordering constraints may be added so that $g_1 < g_3$ holds in the final solution. However, retrieving a case C achieving a goal matching g_2 and no goal matching g_1, is no longer adequate. The reason for this is that in this situation the permutation orders to achieve the goals are either g_3, g_2, g_1 or g_2, g_3, g_1. The resulting order is g_3, g_2, g_1 if g_3 is the only goal achieved in C or if g_2 and g_3 are achieved in C and $g_3 <_C g_2$ holds. The resulting order is g_2, g_3, g_1 if g_2 and g_3 are achieved in C and $g_2 <_C g_3$ holds. In either case, the permutation is no longer $\prec$-consistent as $g3$ is processed before g_1. Because $\prec$-consistent trivial serializability links the order in which the goals must be achieved during the planning process with the order with which the goals must be established in the resulting solution plan, achieving g_3 before g_1 results in a failure. In the example with the domain of process planning, the same situation is encountered if a one-goal case is retrieved achieving U_1. After the case is replayed, the skeletal plan achieves U_1. During the completion process, the skeletal plan must be extended to achieve the other goals; in particular the processing of H. Thus, *processedH1(U_1)* is achieved before *processed(H)*, which is clearly not a $\prec$-consistent permutation to achieve the goals and, as we saw, results in a failure. The order consistency condition is restricted to consider these situations:

Definition 7.5 (Order Consistency Condition Modulo $\prec$) *Suppose that goals in the problem $(I, G, \prec)$ are $\prec$-constrained trivially serializable but not trivially serializable. A case C meets the order consistency condition modulo $\prec$ if there is a substitution θ such that*

1. *$G_C\theta \subset G$.*
2. *For every pair of goals $g, h \in G_C$, if $g\theta \prec h\theta$ holds, then $h <_C g$ must not hold.*
3. *If $g\theta$ is in $G_C\theta \cap G$, then for every* h *in G with* h $\prec g\theta$*, exists* h' *in G_C such that $h'\theta = h$ and h is in $G_C\theta \cap G$.*

Where G_C are the goals achieved in C.

The first two conditions are the requirements of the order consistency condition (Definition 4.6); namely, the goals of the case must match a subset of the goals of the problem and the order of the case must be consistent with the order of the problem. The third condition says that if a goal achieved in the case matches a subgoal $g' = g\theta$ of the problem, all the predecessors of g' relative to $\prec$ must be also matched by goals achieved in the case. This condition ensures that the goals of the problem are solved with a $\prec$-consistent order.

Evaluating the Order Consistency Condition Modulo $\prec$. The GDN allows to test condition 1 and 2 by traversing it. In particular condition 2 is tested based on the following property: the only way that the order consistency condition is not met relative to an ordering $g \prec h$ in the current problem is if g is matched by a goal in a node n_g, h is matched by a goal in a node n_h and n_h is a predecessor of n_g. The algorithm **retrieveCandidateCasesGDN** test this property in the following way (see Figure 4.13): if a goal in a node n matches g with a substitution θ, each predecessor n' of n is tested if it contains a goal matching h with the substitution θ. If any such a node is found, every case indexed below n violates condition 2. Otherwise, each of these cases meets condition 2 relative to $g \prec h$. When this process is made, CAPLAN/CBC simultaneously collects all the goals g' in n' such that $g'\theta \prec g$ holds. In domains in which the conditions of Definition 7.5 must be tested, once these goals are collected and if condition 2 is not violated, CAPLAN/CBC checks if all predecessors of g relative to $\prec$ are also in the collection. If this is the situation, all the cases indexed below n meet condition 3 relative to g. Otherwise, none of them meets it. The same process is repeated if any goal g is matched by the node n currently being revised.

7.4.2 Feature Weighting and $\prec$-Constrained Trivial Serializability

Claim 6.1 states that in domains in which goals are trivially serializable, the context of a feature can be simplified. In particular, goals in the problem that are not matched by goals in the retrieved case C have no influence whatsoever on the outcome of the retrieval. This means that once the skeletal plan has been refitted to a subplan SP achieving $G_C\theta \cap G$, SP can be extended to a solution plan achieving G. We discussed that there are domains in which goals are $\prec$-constrained trivially serializable but not trivially serializable. Suppose that $[g_1,..,g_i, g_{i+1},...,g_n]$ is a $\prec$-consistent permutation of the goals and there is a case C achieving a set of goals G_C and a substitution θ such that $G_C\theta = [g_1,..,g_i]$. In this situation, once the solution of C has been refitted to a subplan achieving $[g_1,..,g_i]$, the subplan can be extended to a solution plan achieving $[g_1,..,g_i, g_{i+1},...,g_n]$. Thus, we can now formulate a restricted form of Claim 6.1:

Claim 7.1 *Suppose that a domain is given in which goals are known to be $\prec$-constrained trivially serializable and not trivially serializable. Then, the factors influencing the effectiveness of the reuse are the initial features of the problem and of the case, the goals common to the problem and the case, and the solution of the case provided that only consistent cases are retrieved.*

As before, retrieval failures still occur if the subplan achieving $G^C \cap G$ in the case cannot be extended to a subplan achieving these goals relative to the initial situation of the new problem. But this claim says that if the subplan can be extended to achieve $G^C \cap G^P$, no failure will occur when extending it further to achieve all the goals in the new problem (i.e., G). This claim is also independent of the particular kind of planner being used.

Chapter 8

Mergeability in Plan-Space Planning

In the previous chapters, the retrieval procedure developed in CAPlan/CbC was presented. It combines dependency-driven retrieval to make a preselection of cases based on the goals and the ordering restrictions $(G, \prec)$ of the current problem and feature weighting to make a final selection based on the initial features I (Chapters 4 and 6). The retrieval procedure selects several cases, each of them covering a disjoint subset of G.

In this thesis, we presented an adaptation method in which cases contain not only the solution plan but also information about the failed attempts to solve the problem. As a result, the adaptation effort is reduced (see Chapter 5). The adaptation method is based on replay in plan-space planning. In this chapter we will study known techniques for merging cases in the context of plan-space planning and based on replay. We will examine the strengths and limitations of these methods and state policies regarding their use.

The first method that we studied follows directly from a general definition of merging as presented in (Kambhampati et al., 1996a). The basic idea is to replay steps of the retrieved cases independent of the steps replayed from the other cases. These independently replayed cases are completed into a solution by the first-principles planner. We call this method *blind merging* as steps are replayed from the current case without considering the steps which have been previously replayed from other cases. A more elaborated merging method, which we call *non-redundant merging*, is to consider the replayed steps to avoid adding steps that already have been introduced. Non-redundant merging was first implemented on a case-based, state-space planner (Veloso, 1994; Veloso, 1997) and later on a case-based, plan-space planner (Ihrig and Kambhampati, 1996a).

Clearly, the use of blind merging results in plans containing redundant steps. That is, the resulting plans contain several steps achieving the same goal. However, it could be possible that the efficiency to find a solution plan is not affected. We carefully examined blind merging based on SNLP and found that it is unlikely that the plan obtained after replaying all the retrieved cases can be extended into a solution plan. This means that steps in the plan obtained after replay will need to be revised to find a solution plan of the current problem. Thus, this method tends to be very inefficient.

Non-redundant merging has been shown to be effective in guiding the planning process of case-based, state-space planners (Veloso, 1994) and of particular interest to our work for case-based, plan-space planners (Ihrig and Kambhampati, 1996a). However, we found two limitations of this method: first, we found that, when using non-redundant merging, requiring that the partial plan obtained to be extensible to a solution plan is a strong requirement (Muñoz-Avila and Weberskirch, 1997a). Second, we will see that in domains having a certain kind of interactions, the size of the partial solution obtained after replay tends to decrease with problem size. Thus, *to retrieve a single case covering much of the goals of the problem or to retrieve fewer cases covering much of the goals is at least equally effective as to retrieve several cases covering all goals in these domains.*

We will also discuss how non-redundant merging is implemented in the context of complete decision replay and how the notions of retrieval failure and beneficial retrieval are extended to multi-case retrieval. Finally, we will re-state the context-simplified claim (see Claim 6.1) for multi-case retrieval.

8.1 Blind Merging of Cases

As a motivation to the discussion that follows, we continue with the example presented in Section 3.5. In this example, the workpiece depicted in Figure 8.1 must be manufactured by considering the cutting tools available. Seven of the areas that must be processed are depicted: the two ascending outlines A_1 and A_2, the horizontal outline H, the two sides $S1$ and $S2$, and the two undercuts U_1 and $U2$. Now suppose that two cases are available in which two plans P_1 and P_2 are contained. Suppose that in $P1$ the areas to the left of the vertical line are processed and that in $P2$ the areas to the right. Thus, for example, the areas A_1, H, U_1, $U2$ and $S1$ are processed in P_1 whereas the areas A_2 and $S2$ are processed in P_2. Suppose in addition that the cutting tools available in the problem were also used to obtain $P1$ and $P2$. In this situation, one would expect that by merging $P1$ and $P2$ a plan to manufacture the workpiece can be easily obtained. We will show, however, that this is not the fact when blind merging is used. But first, we recall the notion of mergeability of plans (Kambhampati et al., 1996a), which as we mentioned is the base of blind merging:

Definition 8.1 (Mergeability of plans) *Given a class* $\mathbf{P}$ *of plans and a plan,* P_1*, in* $\mathbf{P}$ *for achieving a goal* g_1*.* P_1 *is mergeable with respect to a plan* P_2 *in* $\mathbf{P}$ *for achieving a goal* g_2 *if there is a plan* P *in* $\mathbf{P}$ *extending* P_1 *and* P_2 *and achieving both* g_1 *and* g_2*.*

In addition, the plans are said to be **simple mergeable** *if every step in* P *is present in either* P_1 *or* P_2 *and the number of steps in* P *is equal to the number of*

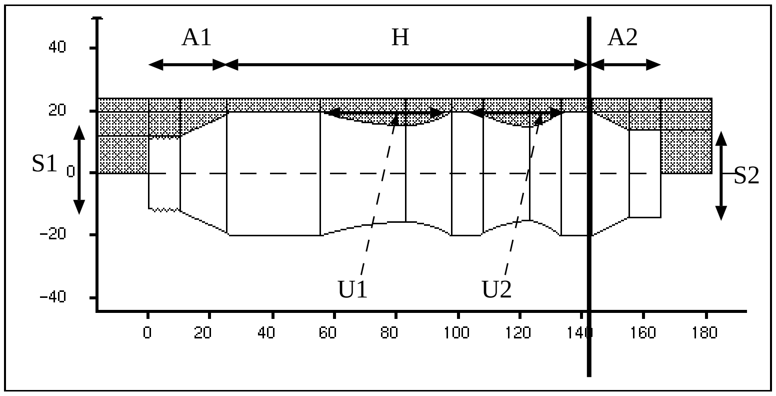


Figure 8.1: Display of a rotational symmetrical workpiece.

steps in P_1 and P_2 (that is, only ordering links are added to extent P_1 and P_2).

As with the definition of serial extensibility, the definition of mergeability does not exclude that backtracking takes place in finding P (see Section 2.4). The point is that no backtracking should take place in the decisions made to obtain P_1 or in P_2. This notion is illustrated in Figure 8.2, where $P1$ and $P2$ are two plans that are extended by adding the step s and four links. The depicted plan does not show that $P1$ and $P2$ are simple mergeable because for that to happen only links can be added to find P. Notice that this definition is only for two plans. The extension for more than two plans is straightforward. The remaining definitions of this chapter are also made for two plans for the sake of simplicity.

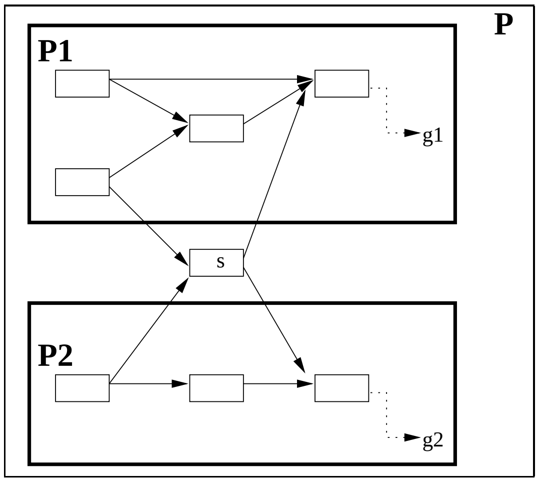


Figure 8.2: Illustration of the notion of mergeability.

In CBP, a maximal gain is expected if a case base is given that is constituted of mergeable plans. If the cases can be replayed totally in the the context of the new problem, the mergeability condition ensures that the planner will not need to revise decisions taken in the cases during the completion phase. It may not always be possible to replay the cases completely. However, if the cases are small in terms of the number of goals solved (e.g., one-goal cases), this situation is likely to occur. The strategy of storing small cases has been explored before to decrease the size of the case base (Ihrig and Kambhampati, 1996a).

Mergeability turns out to be a strong requirement. A first indication of how strong this requirement is can be derived from the following definition:

Definition 8.2 (Parallelizability of Goals) *Given two goals g_1 and g_2. If any two plans P_1 and P_2 achieving g_1 and g_2 are mergeable, then g_1 and g_2 are said to be* **parallelizable**.

If goals in a domain are known to be parallelizable, each goal of a new problem can be solved separately. Afterwards the obtained solution plans can be merged. In (Kambhampati et al., 1996a), however, it is shown that mergeability is a stronger requirement than trivial serializability:

Proposition 8.1 *If a set of goals is parallelizable, the goals are trivially serializable.*

The opposite direction does not hold; goals might be trivially serializable but not mergeable (an example will be shown later in this section). Still, that does not say anything with respect to the example with the workpiece domain presented in this section; even if the seven goals are not parallelizable, it could be that the plans P_1 and P_2 are mergeable. Non-parallelizability means that there are plans achieving the goals that are not mergeable. Thus, if the goals are not parallelizable does not necessarily means that all plans are nonmergeable. The following proposition explains why the plans P_1 and P_2 processing parts of the workpiece depicted in Figure 8.1 are not mergeable:

Proposition 8.2 *Suppose that first-principles planning is done with SNLP. Then, a complete plan P_1 for achieving a goal g_1 is mergeable with respect to a complete plan P_2 for achieving a goal g_2 if and only if P_1 and P_2 are simple mergeable.*

Proof. Clearly, if P_1 simple mergeable with respect to P_2 relative to g_1 and $g2$, then these plans must also be mergeable relative to these goals. To show the opposite direction, suppose by contradiction that a plan step s was added by SNLP to extend P_1 and P_2. New steps are added by SNLP only to achieve open preconditions. Thus, either P_1 or P_2 have an open precondition. This is a contradiction because P_1 and P_2 are complete. Thus, no step could have been added to find P. As a result, the plans must be simple mergeable relative to g_1 and g_2. ■

Although the proof of this proposition is simple, it has a significant consequence: if for extending two plans to a solution plan any step must be added, backtracking will take place because the plans are nonmergeable. For example, in the domain of process planning there are always interactions (threats) between any two complete plans because the use of the cutting tools and the clamping operations must be rationalized. Concretely, every step holding a tool in a plan will interact with every step holding a tool in the other plan. Theoretically, to resolve these interactions (conflicts), it is enough to add links. However, in the specification developed (see Appendix A), not only ordering links must be added between the holding steps to solve the threats, but new steps must be introduced as well for performing the operation that unmounts the tool from the holding machine, *MakeToolHolderFree*. This is not particular to our specification; the same situation occurs if the specification of (Gil, 1991) is used (the unmount operation is called *Release-from-holding-device* there). As a result, proposition 8.2 shows that machining plans are not mergeable for SNLP if either of these specifications is used.[1] In particular the two plans processing the two parts of the workpiece depicted in Figure 8.1 are not mergeable. Thus, backtracking will take place when merging them.

Example of Goals that are Trivially Serializable but Not Mergeable. For another example consider the initial situation in the logistics transportation domain depicted in Figure 8.3: there are three post offices A, B and C, two packages p_1 and p_2 and a truck. The packages and the truck are initially located in B. Suppose that the goals of the problem are (1) to relocate p_1 in A and (2) to relocate p_2 in C. The arcs 1 and 2 depict two subplans achieving the two goals separately; namely, the first goal is achieved by loading p_1 in the truck, moving the truck from A to B and then unloading p_1 from the truck. The second plan follows the same scheme: p_2 is loaded in the truck, the truck is moved from B to C and then p_2 is unloaded. The point here is that there is no possible extension of these subplans without adding a plan step; independent of how SNLP orders the steps of the subplans, a new step must be added. Notice first that the subplans must be ordered so that one is executed before the other one because they use the same resource (i.e., the truck). If, for example, the first subplan is ordered before the second one, a step must be added between them moving the truck back to B so that the second plan can be performed (this movement is depicted with the doted line). Proposition 8.2 says that these two subplans cannot be mergeable in SNLP. Goals in the logistics transportation domain are trivially serializable for all classes of plans. The reason for this, is the reversible character of its operations. For example, trucks can always be moved back to its original destination.[2] Thus, these two goals are an example of goals that are trivially serializable but not mergeable.

[1] In (Muñoz-Avila and Weberskirch, 1996c) an explanation of why it is natural to model the holding operations in this way is given.

[2] This, however does not mean that problems in this domain are easy. The computational effort to extend a particular plan can be very large.

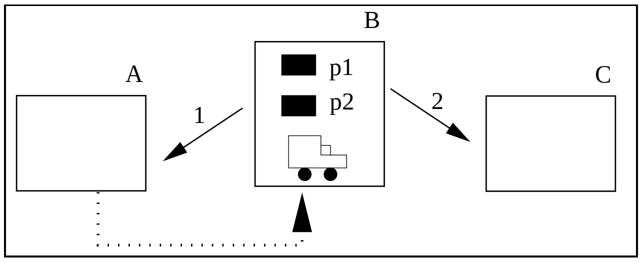


Figure 8.3: A situation in the logistics transportation domain

Meaning of Proposition 8.2 for CBP. The meaning of this result for CBP is also significant: if some of the retrieved cases can be replayed completely and steps must be added to extend them, the retrieval will fail because replayed steps must be revised. This is a sensitive issue, in particular for case-based planners based on SNLP having as storage policy to store only cases achieving fewer goals as possible (Ihrig and Kambhampati, 1996a). If a multi-goal problem is given, it is likely that some of these cases can be completely replayed but they cannot be merged. Even for other approaches, the result shows the strong limitations of blind merging based on SNLP because precisely in the situations where a maximal gain is expected from CBP, namely, when the description of the retrieved cases are entirely subsumed in the description of the current problem, a retrieval failure occurs. In this situation, blind merging will be able to replay the whole cases but the SNLP can only extend them if and only if only ordering constraints need to be added. In Chapter 9 it will be shown that in general blind merging is in fact very innefficient and is outperformed even by first-principles planning.

8.2 Non-Redundant Merging of Cases

A disadvantage of blind merging is that the solution obtained may be very large because several steps are repeated unnecessarily. To avoid redundancy, another form of merging, which we call *non-redundant merging*, has been proposed (Veloso, 1994; Ihrig and Kambhampati, 1996a). During the replay phase, opportunities to establish preconditions are considered in the following way: before replaying a step to establish an open precondition, the system checks if the precondition can be established with a step in the current subplan (i.e., the subplan already obtained from the previously replayed cases). If this is possible, the step is not replayed and the precondition is left open. During the completion process, the first-principles planner prefers to establish the conditions left open by the replay process by using the available steps. That is, the first-principles planner prefers to perform simple establishments than establishments with a new plan step. New steps are added only if no completion of the plan by using the steps available is possible. If more than two cases are involved, the process is done stepwise: the first two are merged and extended to obtain a complete subplan (i.e., a subplan containing no open preconditions and no threats). The third case is then merged with the current subplan and so on. The rationale behind non-redundant merging is to take advantage of opportunities to establish the open preconditions using steps that were not available when each case was solved because they were solved separately.

Example of the Effect of the Interactions in Merging. The way the interactions between the goals affect the non-redundant merging process is illustrated with the example in Figure 8.4. *cs*, *hs* and *ms* denote clamping, holding and machining steps respectively. That is, steps to mount the workpiece, steps to hold a cutting tool and steps to process an area. This figure depictes two subplans achieving *machined(H)* and *machinedH1*(U_1), processing the area H and half of the area U_1 of the workpiece depicted in Figure 8.1. The partial plan is obtained with non-redundant merging and when the subplan achieving *machined(H)* is generated before the subplan achieving *machinedH1*(U_1). A processing step *cs(left)*, depicted with a dashed box, was not replayed because another step *cs(left)* in the other subplan establishes the precondition *clamped(left)* (the double arrow represents a threat that would occur if this step had been replayed). As a result, the part of the case achieving the preconditions of *cs(left)* is not replayed, too.

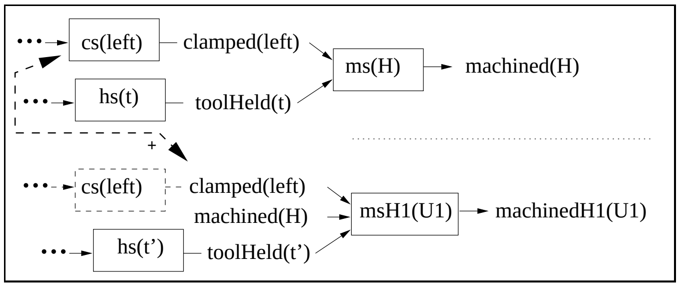


Figure 8.4: Interactions between two subplans.

In the previous section we discussed Kambhampati's result which shows that parallelizability is an stronger requirement than trivial serializability. We will now formalize the notion of non-redundant mergeability to analize this merging method.

Definition 8.3 (Non-Redundant Mergeability, Parallelizability) *Given a class* $\mathbf{P}$ *of plans and two plans* P_1 *and* P_2 *in* $\mathbf{P}$, *then* $P_2 \setminus P_1$ *denotes the plan that remains after removing*

1. *causal links from* P_2 *achieving preconditions that can be established by using steps in* P_1, *and*
2. *plan steps in* P_2 *that were added as the source of a removed causal link, and*
3. *causal links achieving preconditions of removed plan steps.*

Let g_1, g_2 two goals achieved by P_1 and P_2. P_1 is **non-redundant mergeable** *with respect to a plan P_2 if there is a plan P in $\mathbf{P}$ extending P_1 and $P_2 \setminus P_1$ achieving both g_1 and g_2.*

Given two goals g_1 and g_2. If any two plans P_1 and P_2 achieving g_1 and g_2 are non-redundant mergeable, then g_1 and g_2 are said to be **non-redundant parallelizable**.

In other words P_1 and P_2 are non-redundant mergeable if P_1 and $P_2 \setminus P_1$ are mergeable relative to g_1, g_2 and $\mathbf{P}$. Thus, the mergeability of P_1 and P_2 implies their non-redundant mergeability relative to g_1, g_2 and $\mathbf{P}$: if P_1 and P_2 are mergeable, there is a plan P in $\mathbf{P}$ extending P_1 and P_2. To extend P_1 and $P_2 \setminus P_1$, the latter is extended to P_2 by adding the pruned steps. Then, P_1 and P_2 are extended to P.

Example showing that Non-Redundant Mergeability does not imply Mergeability. The example depicted in Figure 8.5 shows an example of a situation in the logistics transportation domain were two subplans are non-redundant mergeable but not mergeable. There are three post offices A, B and C. In addition two packages, p_1 and p_3, are located in A, other two packages, p_2 and p_4, are located in B and a truck is located in A (not shown in Figure 8.5 (a)). Suppose that two subplans are available, P_1 and P_2. In the plan P_1, (1) p_1 is loaded in the truck, (2) the truck is moved from A to B, (3) $p2$ is loaded in the truck, (4) the truck is moved from B to C and (5) the two packages are downloaded. The path followed by the truck is marked by the arcs 1 and 2. The fragments (4) and (5) are shown in Figure 8.5 (b). The subplan $P2$ relocates the packages p_3 and p_4 by following the path marked by the arcs 3 and 4 (again the last fragments of this subplan are depicted in Figure 8.5 (b)). P_1 and P_2 are not mergeable because to extend them a step moving the truck from C to A must be added; if, for example, P_1 is ordered before $P2$, then once the truck has to moved back to A after p_1 and p_2 are unloaded in C. P_1 and P_2 are non-redundant mergeable because $P_2 \setminus P_1$ consists only of the two unloading steps(i.e., *unload(p3,tr)* and *unload(p4,tr)*). The other steps are removed. For example, the step *move(tr,B,C)* of $P2$ is removed because the preconditions of *unload(p3,tr)* and *unload(p4,tr)* can be achieved with an step in $P1$ (i.e., *move(tr,B,C)*). Any solution plan relocating $p3$ and $p4$ necessarily contains the two unloading steps. Thus, P_1 and $P_2 \setminus P_1$ are mergeable.

The previous example shows that non-redundant mergeability is a weaker requirement than mergeability. To continue our analysis a new definition is introduced:

Definition 8.4 (Non-Redundant Parellelizability of Goals) *Given two goals g_1 and g_2. If any two plans P_1 and P_2 achieving g_1 and g_2 are non-redundant mergeable, then g_1 and g_2 are said to be* **non-redundant parallelizable**.

Even though it is a weaker requirement than mergeability, non-redundant parallelizable is still stronger than trivially serializable (compare to Proposition 8.1):

Proposition 8.3 *If a set of goals g_1, ..., g_n is non-redundant parallelizable, then it is trivially serializable.*

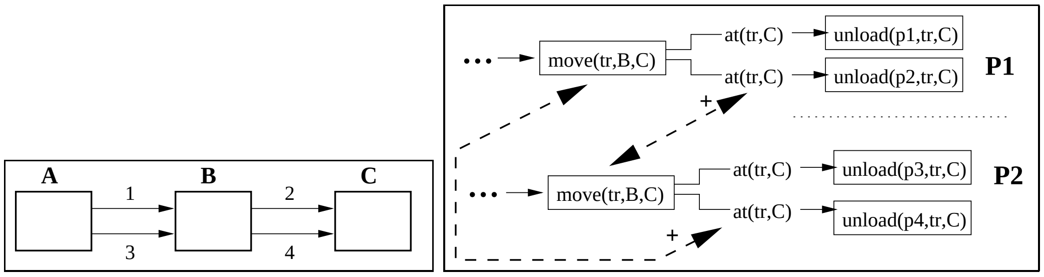


Figure 8.5: Example of a situation in the logistics transportation domain with (a) the path followed by the two subplans and (b) two subplan fragments.

Proof. We will prove this result for two goals. Let P_1 be a plan achieving g_1. We will show that P_1 can be extended to a plan achieving g_1 and g_2. Let P_2 be any plan achieving g_2 and SEQ_2 the sequence of planning decisions taken to obtain P_2. Let $SEQ_{P_2 \searrow P_1}$ be the subsequence of planning decisions in SEQ_2 achieving $P_2 \searrow P_1$. Because g_1 and g_2 are non-redundantly parallelizable, there must be a plan P_3 extending P_1 and $P_2 \searrow P_1$ that achieves g_1 and g_2. Let SEQ_3 be the sequence of planning decisions to obtain P_3 by extending P_1 and $P_2 \searrow P_1$. Then, P_1 can be extended to a plan achieving g_1 and g_2 by following the sequences of planning decisions $SEQ_{P_2 \searrow P_1}$ and then SEQ_3. This shows that g_1, g_2 is a serialization order. In the same way it can be shown that g_2, g_1 is also a serializiation order. Thus, g_1, g_2 are trivially serializable. ■

8.3 Positive Interactions and Non-Redundant Merging

Continuing with our study of the non-redundant merging method, notice that in the two examples discussed in the previous section and depicted in Figures 8.4 and 8.5, no *positive* threats between P_1 and $P_2 \searrow P_1$ occur. This is not a coincidence: positive threats indicate that the same effect has been obtained in P_1 and P_2, which is the kind of redundancy that is eliminated for constructing $P_2 \searrow P_1$. In the example in Figure 8.4, one of the subgoals, *clamped(left)@msH1(*U_1*)*, of the machining subplan achieving the goal *machinedH1(*U_1*)* is left open and as a result a significant portion of P_2 is removed. Notice that if *cs(left)* would have been added to achieve *clamped(left)@msH1(*U_1*)*, the positive threat *cs(left)* $\overset{+}{\longleftrightarrow}$ *(cs(left)* $\rightarrow$ *clamped(left)@msH1(*U_1*))* would have occur. In the example in Figure 8.5, all steps but two are removed from the P_2. Again, two positive threats are shown, *move(tr,B,C)* $\overset{+}{\longleftrightarrow}$ *(move(tr,B,C)* $\rightarrow$ *at(tr,C)@unload(p2,tr,C))* and *move(tr,B,C)* $\overset{+}{\longleftrightarrow}$ *(move(tr,B,C)* $\rightarrow$ *at(tr,C)@unload(p4,tr,C))*, which would have occur if the corresponding steps would have been added. This illustrates the following claim:

Claim 8.1 *the more positive threats occur between P_1 and P_2, the smaller $P_2 \setminus P_1$ is.*

In the logistics transportation domain, the occurence of threats depends on the particular situation. In the example depicted in Figure 8.5, several positive threats occur because of the use of the truck. However, if another truck is available and $P1$ uses one truck and P_2 the other one, no positive threats occurs. In the domain of process planning the situation is very different: threats always occur between any two subplans; these threats indicate violations in the use of the clamping and holding operations.[3] Several threats will occur between any two processing subplans P_1 and P_2; namely, two for each pair of clamping or holding steps such that one is in P_1 and the other one is in P_2. Further, any complete manufacturing subplan contains at least one clamping and one holding step because to process any area, the workpiece must be clamped from a certain position and a certain tool must be held. The more threats occur, the more likely it is that positive threats occur because the number of clamping and holding operations is limited. In our specification, six different clamping operations are possible, but some of them may not be applicable depending on the particular workpiece being manufactured.

Claim 8.1 is significant for CBP: if in a domain, positive threats are likely to occur between subplans, retrieving several cases is not an adequate strategy. As more cases are replayed with the non-redundant merging method, less guidance is provided by replaying these cases. The reason for the decrease in the guidance is that more steps tend to be discarded as more positive threats occur between the subplan obtained after replaying previous cases and the subplan obtained after replaying the current case. The following claim summarizes this observation and will be supported in Chapter 9, where several experiments in different domains were performed.

Claim 8.2 *In domains where positive threats frequently occur between subplans, retrieving a single case or fewer cases covering much of the goals is a more effective retrieval strategy than retrieving several cases covering all the goals.*

8.4 Merging with Complete Decision Replay

In complete decision replay the justifications are reconstructed to reduce the completion effort of the skeletal plan by discarding completion possibilities that are know from the case to fail (see Section 5.6). A failed decision in the case is considered as failure in the new situation only if its justification $\{a_1, ..., a_n\}$ can be reconstructed.

[3]At any time of the manufacturing process, at most, one cutting tool can be held and one clamping operation can be performed.

That is, only if for each assignment a_i, the decision d_i containing it has been replayed. This means that when merging several cases, the justifications reconstruction process can be performed immediately after each case is replayed, which is what CAPLAN/CBC does. Replaying additional cases does not affect the invalidity of a decision d as its invalidity is determined by the set of decisions $\{d_1, ..., d_n\}$ in the case. Whether these decisions will not need to be revised during the merging process depends on the particular situation. If any of these decisions needs to be revised, the invalidity of d is revised. But, this is precisely the kind of contingencies that REDUX (and thus CAPLAN) is capable of handling (see Chapter 5).

Clearly, the number of justifications reconstructed depends on the number of steps of the current case that were replayed. In domains where positive threats occur frequently between subplans, fewer justifications are reconstructed from the current case as more cases have been previously replayed to solve the current problem. The reason for this is that the less decisions are replayed, the less likely is that the justifications $\{a_1, ..., a_n\}$ can be reconstructed because the assignments a_i can only be reconstructed if their corresponding decisions d_i are replayed (see Section 5.6). This means that in these domains not only the guidance provided by replay tends to reduce with the number of cases being replayed but also the guidance provided by complete decision replay. This also supports Claim 8.2.

8.5 Multi-Case Retrieval and Merging

In this section, we will extend the notions of retrieval failure and beneficial retrieval for multiple cases (see Definitions 6.3 and 7.2). We will also re-state the context-simplified claim (see Claim 6.1). The notions are based on the non-redundant merging method. We will make no further comments about the meaning of these concepts as they were discussed in detail in Sections 6.2, 7.2 and 6.3. First, we state the notion of retrieval failure:

Definition 8.5 (Retrieval Failure, Adequate Retrieval) *Given a problem P, the retrieval of C_1 and C_2 is adequate if their corresponding plans Pl_1 and Pl_2 are non-redundant mergeable. Otherwise, the retrieval is a failure.*

The notion of beneficial retrieval for multiple cases is stated as follows:

Definition 8.6 (Beneficial Retrieval) *Given a plan Pl,* searchSpace(P) *counts the number of valid and failed decisions made when generating Pl.*

Given a solution plan Sol of a problem P obtained by the non-redundant merging of C_1 and C_2, then the retrieval of C_1 and C_2 is beneficial with respect to P and C if:

$$searchSpace(Sol)/(searchSpace(Skel_1) + searchSpace(Skel_2 \setminus Skel_1)) \leq thr_{ben}$$

where $Skel_1$ is the skeletal plan obtained after replaying C_1 and $Skel_2$ is the skeletal plan obtained after replaying C_2. thr_{ben} is a predefined threshold, called the benefit threshold.

Finally, we closed this section by stating the context-simplified claim for multi-case retrieval.

> **Claim 8.3 (Context-Simplified)** *In domains where goals are non-redundant parallelizable, the feature context are the initial features of the problem and of the cases, the goals common to the problem and the cases, and the solution of the cases.*

Chapter 9

Empirical Validation

In the previous chapters, the twofold retrieval process in CAPLAN/CBC has been presented (Chapters 4 and 6) and complete decision replay, an extension of standard replay for plan-space planners, has been discussed (Chapter 5). In addition, integration issues in CAPLAN/CBC were studied (Chapter 7) and the scope of plan mergeability was presented (Chapter 8). In this chapter we will examine the performance of these methods. More concrete, the following issues will be studied.

Performance of Dependency-driven Retrieval. Dependency-driven retrieval will be compared against goal-driven retrieval. The performance of the retrieval process and the problem-solving process will be measured when both methods are used. We will also measure the performance of retrieval with the GDN when no ordering restrictions are given. That is, when goal-driven retrieval is performed but the indexing structure is the GDN.

Performance of Dynamic Retrieval based on Feature Weighting. The increase in the reliability of the retrieval when using feature weighting will be measured. In addition, the policy to create cases based on the retrieval benefits is compared against the policy to create cases based on the retrieval failures.

Performance of Complete Decision Replay. The performance of complete decision replay is compared against the performance of standard replay. The effectiveness of the dual integration between retrieval and adaptation will also be measured.

Performance of the Overall Case-Based, Problem-Solving Process in CAPLAN/CBC. The performance of CAPLAN/CBC will be compared against first-principles planning with CAPLAN (SNLP). Even though previous research has shown already that case-based planning outperforms first-principles planning (Veloso, 1994; Kambhampati, 1994; Koehler, 1994; Francis and Ram, 1995b; Bergmann and Wilke, 1995a; Ihrig and Kambhampati, 1996a), we perform this experiments to confirm that not only the techniques for CBP presented in this thesis improves the efficiency over previous techniques, but to show that the combination of these techniques in CAPLAN/CBC results in a powerful system.

Performance of the Merging Methods. The claims about the ineffectiveness of blind merging in plan-space planning will be confirmed experimentally. In addition, Claim 8.2, regarding the limitations of non-redundant merging in domains where positive interactions frequenly occur, will be corroborated experimentally.

9.1 Problem Domains

We performed experiments with the domain of process planning (see Appendix A), the artificial domain ART-1D-RD-RES (see Appendix C), the logistics transportation domain (see Appendix B) and the extension of the logistics transportation domain presented in Section 6.3. As discussed before, the domain of process planning is characterized by the high number of interactions between subplans achieving goals corresponding to processing areas of a workpiece (see, Section 8.3). The source of the interactions are restrictions on the use of manufacturing resources; namely, the cutting tools and the clamping machine.

The artificial domain ART-1D-RD-RES is an extension of the domain ART-1D-RD (Kambhampati, 1993); two new operators were added that rationalize the use of the resources *he*, renamed *occ-a*, and *hf*, renamed *occ-b*, instead of adding and deleting them directly in the actions A_i. In this way, any problem solver should be confronted to similar situations as with the domain of process planning. In this domain, valid ordering restrictions can be predefined; namely, the goal G_i should always be achieved before the goal G_{i+1} for any i. In addition, there are always interactions between any two subplans achieving G_i and G_j.

The third domain we used in the experiments is the logistics transportation domain, originally specified in (Veloso, 1994). In this domain, there are also resources; namely, trucks and airplanes. However, in contrast to the domain of process planning, the more resources are made available, the more likely is that subplans achieving goals are independent. That is, no interactions necessarily occur between the subplans. For example, if two packages must be relocated within the same city and two trucks are available, each truck can be used to relocate one package.

The extension of the logistics transportation domain adds the following restrictions: trucks must not be moved into the same post office more than once and problem-specific restrictions such as not allowing the truck to move from a certain post office to another post office. As a result of these restrictions the goals may not be trivially serializable.

9.2 Performance of Dependency-Driven Retrieval

To measure the effectiveness of dependency-driven retrieval we compare it against goal-driven retrieval. More concrete, the retrieval was made by comparing the performance of the case-based, problem solving process when the retrieved cases meet the ordering consistent condition against cases meeting just condition 1 of the ordering consistent condition (see Definition 4.6). To isolate the effects of using the dependency-driven retrieval technique, the feature weighting mechanism was disable in this experiment. All relevant features were given a weight of 1, which remained invariant throughout the experiment. For the same reason, standard replay was used instead of complete decision replay. The retrieval threshold was set to 75% (see Definition 6.2).

9.2.1 Experiment Setup

The experiment was performed in the domain of process planning and in the artificial domain. The ordering restrictions in the domain of process planning were computed by the geometrical reasoner. The experiment setup consisted of an ordered collection of problems of ascending goal size; five problems consisting of one goal, ten problems consisting of two goals, ten problems consisting of three goals and so on until ten problems consisting of eight goals. To narrow the spectrum of possible problems, a subset of types of processing areas was randomly selected.

The ordering restrictions in the artificial domain were stated as $G_i \prec G_j$ for every $i < j$. As with the domain of process planning, the collection was ordered in sequences of problems consisting of ascending number of goals; namely, five problems consisting of a single goal and the rest of sequences consisting of 10 goals (one for two, three and so on until eight goals). To narrow the spectrum of possible problems, ten goals were preselected, G_1, ..., G_{10}.

For each domain, two case bases were constructed, the three-level case base of CAPLAN/CBC, in which the main discrimination criterion between the cases are the dependencies between the goals, and the case base of PRODIGY/ANALOGY, in which the main discrimination criterion are the goals. All basic operations such as matching and the representation of basic data types such as collections of predicates are common to both architectures. In this way, none of them takes advantage on the other one because of implementation details.

Initially, both case bases were empty. Each time a problem was solved, the problem and the found solution were added as a new case to each case base. To solve a problem, a case was retrieved and adapted. We added the retrieval and adaptation times to measure the performance of the overall case-based, problem-solving process. Notice that, for a given problem, if a case is retrieved with the dependency-driven retrieval technique, a case will be retrieved with the goal-driven retrieval. The opposite, however, does not hold. If no case was retrieved, we added the time required by the retrieval procedure to find out that no case was available to

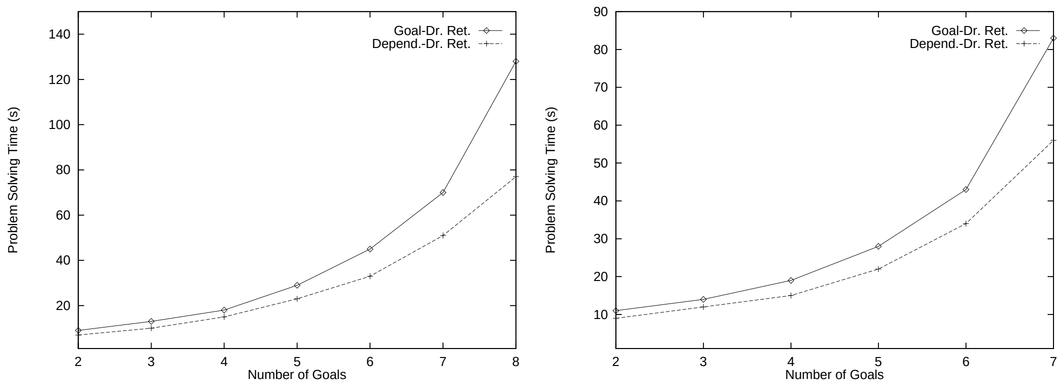


Figure 9.1: Problem solving times with dependency-driven retrieval and goal-driven retrieval for (a) the domain of process planning and (b) the artificial domain.

the time required by the base-level planner CAPLAN to solve the problem.

9.2.2 Results of the Experiment with Dependency-Driven Retrieval

Figure 9.1 depictes the results. The overall problem-solving time for the problems consisting of i goals ($i = 1, 2, ...$). This results shows a significant improve in the performance as a result of the dependency-driven retrieval technique. They also show the importance of considering the goal orderings in domains where such information is available.

Retrieval times. In this experiment, we measured the retrieval times. In addition to the retrieval time with the dependency-driven and the goal-driven techniques, we measured the retrieval time when no ordering restrictions were given but the GDN was used to index the cases. That is, when goal-driven retrieval is performed and the GDN is used as the indexing structure. As discussed in Section 4.8, a decrease in performance is expected when no ordering constraints are given because in the GDN the same set of goals can be repeated several times. The results are depicted in Figure 9.2. The retrieval time when no ordering restrictions were given was quite high in comparision to the retrieval time when the ordering restrictions were given. This shows that using the GDN in domains where no ordering restrictions can be predetermined is an inadequate strategy. On the other hand, these results also show that not only the overall case-based, problem-solving process is more efficient with dependency-driven retrieval, but, particularly, that the retrieval time also decreases provided that the ordering constraints are given.

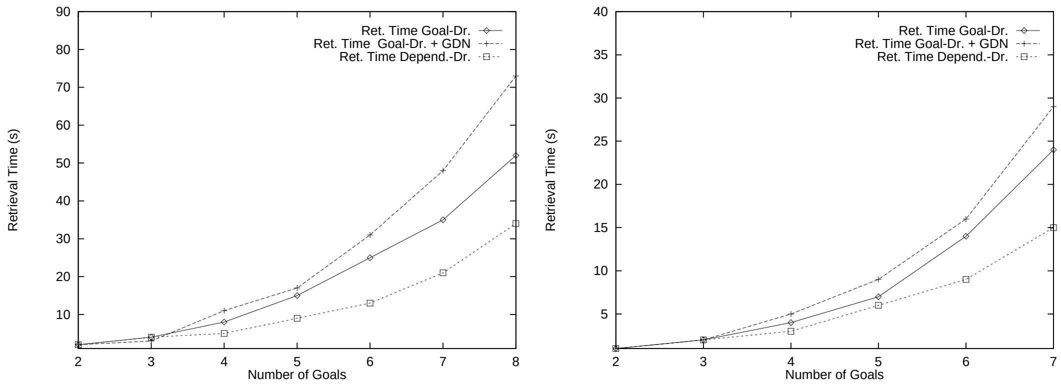


Figure 9.2: Retrieval times by performing goal-driven retrieval and using the GDN as indexing structure, by performing dependency-driven retrieval and by performing goal-driven retrieval. These time measures are depicted for (a) the domain of process planning and (b) the artificial domain.

9.3 Performance of Feature Weighting in CBP

In this section a report will be made on experiments performed in which the increase in the reliability of the retrieval by using feature weighting was evaluated. To perform this evaluation, retrieval was made in a problem-solving cycle also including adaptation and learning. Retrieval was performed in two modes, dynamic and static. In the dynamic mode, the weighted similarity condition was used in the framework presented in Chapter 6. That is, the feedback of the problem solver indicating if the retrieved cases failed or not was considered and the weights of the filtered features were updated. In the static mode, the weights of the relevant features were always set to one and they remain unchanged throughout the experiment.

9.3.1 Learning Weights on Single Cases

In the first experiment, the effect of feature weighting in single cases was studied. The purpose of this experiment is to measure how the feature weighting process affects the reliability of the retireval for single cases.

This experiment consisted of two parts: in the first part, the experiment was made on domains meeting the conditions of Claims 6.1 and 7.4.2. In the second part, the experiment was made in a domain not meeting these conditions. The experiments for the first part were performed in the domain of process planning (with ordering restrictions) and in the logistics transportation domain. The second experiment was made in the extension of the logistics transportation domain. Adaptation was made with complete decision replay though it does not play any role in the results because we measured the increase in reliability and not the time performance..

Experiments in Domains meeting Claims 6.1 and 7.4.2

Experiment Setup. The experiment setup consisted of 5 runs. In each run, a problem, called the pivot problem, was stated. A solution for the pivot problem was found; the solution together with the problem and the relevant features were used to form a case, C. All feature weights of C were set to 1 and k_C, f_C were set to 0. Then, some features of the pivot problem were randomly fixed. A new goal and new features that do not occur in the pivot problem were also given. Taking as basis the pivot problem, new problems were formed by changing the fixed features, or/and by adding the new goal and the new features. Changing a fixed feature means changing the relations between the objects mentioned in the feature. For example, if a feature states that a truck is in a certain location, the changed feature will state that the truck is in another location. The problem collection met the following conditions:

1. For every problem in the collection and if the weights of all features is set to 1, C meets the retrieval condition with the retrieval threshold set to 75% (see Definition 6.2).

2. The number of times that fixed features were changed in the collection is the same. For example, if a fixed feature indicates the location of a truck and another fixed feature indicates that a post office is in a certain city, the number of problems in which the truck is changed of location is the same as the number of problems in which the post office is changed of city.

3. If n denotes the number of fixed features, then problems were ordered in a way that within a sequence of problems, $Problem_{mn+1}, ..., Problem_{mn+n}$, the number of changes of a fixed feature is the same ($m = 0, 1, ...$). For this reason, the number of problems in the collection is a multiple of the number of selected features.

In the experiments the multiple factor was 5. In addition, in the logistics transportation domain 5 features were fixed and in the domain of process planning 6. Thus, the collections consisted of 25 problems in the first domain and 30 in the second one. The total number of problems involved were 125 in the logistics transportation domain and 150 in the domain of process planning.

Discussion about the Experimental Setup. The ideal experiment to show the increase of reliability with feature weighting is to form all possible combinations of colections of problems and show that the increase occurs in average. Because such a process implies a combinatorial explosion, we stated conditions (2) and (3) to equally distributing the effect of every change in the fixed features and of capturing the average situation. Condition (2) ensures that no feature takes advantage of the other ones by changing them the same number of times. As discussed in Chapter 6, the relation of f_C to k_C determines the incremental rate of the feature weights.

Thus, if not matching a feature causes a retrieval failure, the changes of weights will be greater when this feature is changed in problems allocated at the beginning of the collection. In contrast, if these problems are allocated at the end of the collection, the changes in the feature weights are reduced. For this reason, condition (3) ensures that the final weight is closer to the average by distributing the problems changing the feature equally through the collection.

Results. Tables 9.1 and 9.2 summarize the results of this experiment for the domain of process planning and for logistics transportation domain respectively. The first row of these tables presents the percentage of times the pivot cases was retrieved. The second row presents the percentage of times that retrieving the pivot cases resulted in a retrieval failure. Each of the first five columns presents the results for a sequence $Problem_{mn+1}, ..., Problem_{mn+n}$ in the collection (m = *1, 2, ..., 5*). The sixth column shows the results for the whole collection when the cases are retrieved in static mode.

	Dynamic					**Static**
Items	**1**	**2**	**3**	**4**	**5**	
% Cases Retr.	82	71	63	51	49	100
% Retr. Failures	41	26	19	7	4	47

Table 9.1: Measures of the reliability of the retrieval in the domain of process planning by weighting features and without weighting features.

	Dynamic					**Static**
Items	**1**	**2**	**3**	**4**	**5**	
% Cases Retr.	94	84	73	65	57	100
% Retr. Failures	37	27	15	9	6	41

Table 9.2: Measures of the reliability of the retrieval in the logistics transportation domain by weighting features and without weighting features.

Discussion of the Experiment Results. These experiments show that feature weighting increases the reliability of the retrieval by decreasing the percentage of retrieval failures. This can be seen by comparing the fifth column against the first column of each table. That is, comparing the reliability of retrieval with the dynamic mode after several retrieval episodes against the reliability of retrieval in the static mode. Notice, in addition, that in the later runs the changes in the percentages tends to decrease. This shows that there is a tendency of the feature weights to converge in the average situation constructed in the experiment setup. As discussed in Section 6.5, a problem of feature weighting is that cases may become specialized.

	Static	**Dynamic**	
		noEBL	**EBL**
% Cases Retrieved	100	74	57
% Retrieval Failures	56	42	18

Table 9.3: Measuring the reliability of the retrieval in a general domain.

In the domain of process planning and in the last run, from the 49% of the cases retrieved, 4% were failures. This means that effectively 45% of the retrieval episodes were adequate. The percentage of situations in which the retrieval is correct is 53% for all the problems (i.e., 100% - 47%). This means that the specialization of the cases results in the incorrect exclusion of 8% the problems. This percentage, however, is small compared to the gains in reliability of the retrieval reducing from 47% of the retrieval failures to only 4%. A similar observation can be made for the logistics transportation domain.

Experiments in a General Domain

The domain used in this experiment is the extension of the logistics transportation domain, which does not meet the conditions of Claim 6.1 (see Section 6.3).

Experiment Setup. The experimental setup was the same as described in the previous experiments; namely, five runs were made. In each run, a problem was fixed and a solution was found to form the pivot case. A collection of problems was made by changing features in fixed problem and distributing these changes equally throughout the collection. 4/5 of each collection was used as training examples and the results were measured for the last 1/5 of the examples. Each case was trained in two modes: the *noEBL-mode* and the *EBL-mode*. In both modes feature weights are updated. In the EBL-mode and if the retrieval failed, a test is made to detect if the failure was caused by the goals interacting negatively with the case (see Section 6.4). If this is the situation, the weights are not updated. In contrast, in the noEBL-mode feature weights are always updated.

Results. The results are shown in table 9.3. The first column shows the results when weights were not considered for retrieval (i.e., the weighted retrieval condition was tested with all weights set to 1). The second column shows the results when feature weights were updated in noEBL-mode. The last column shows the results when feature weights were updated in EBL-mode. The first row shows the percentage of cases retrieved and the second one the percentage of retrieval failures.

Discussion of the Experiment Results. Notice that the best results are obtained with the EBL-mode, where the percentage of retrieval failures was reduced from 56% (with the static mode) to 18%. The percentage of retrieval failures cannot be further reduced because of the negative interactions of the goals that are not in the case. Still, the improve in the reliability is significant relative to the static mode. With the noEBL-mode, the percentage of retrieval failures also decreases from 56% to 42% although not as significant as with the EBL-mode. A negative effect is the specialization of the cases in the noEBL-mode; namely, 32% of the retrieved cases were correct (74% - 42%). Compared to the 44% of the static mode (100% - 56%), this means a specialization factor of 12% (44% - 32%). In the EBL-mode the specialziation factor is just 5% (44% - 39%). The difference in results between the noEBL and the EBL modes shows the importance of censoring the learning method in general domains.

9.3.2 Policies to Create New Cases

In CAPLAN/CBC new cases are created only if the adaptation effort of the retrieved cases is considered nonbeneficial (see Section 7.2). Previous approaches to CBP add new cases either eagerly (i.e., every found solution is added as a new case) or if a retrieval failure occurs (see Definition 6.3). In parallel to the experiments performed in Section 9.3.1 showing the increase in the reliability of the retrieval by using feature weighting, we evaluated the policies to create new cases. The beneficial threshold was set to 2 (see Definition 6.3) and the following items were measured:

1. Percentage of cases retrieved (already reported in Tables 9.1 and 9.2).
2. Percentage of retrieval failures (already reported in Tables 9.1 and 9.2).
3. Percentage of nonbeneficial retrievals.
4. Percentage of case-based, problem-solving episodes in which the retrieval is a failure but beneficial.
5. Percentage of case-based, problem-solving episodes in which the retrieval is adequate but nonbeneficial.

Results. The results are summarized in tables 9.4 and 9.5. Each row shows each of the five items in the same order as listed before. Each column *i* averages the results for the *i*-th sequence of each run. The sixth column shows the results for the whole collection with the static retrieval mode.

Discussion of the Experiment Results. From these results, some observations can be made. First, even in the last sequences, when the percentage of retrieval failures decreases, nonbeneficial retrieval episodes are still likely to occur. Thus, the adaptation effort can be significant independent of the fact that the retrieval tends to

	Dynamic					**Static**
Items	**1**	**2**	**3**	**4**	**5**	
% Cases Retr.	82	71	63	51	49	100
% Retr. Failures	41	26	19	7	4	47
% Nonben. Retr.	15	12	17	11	15	24
% Fail. & Ben.	8	6	3	2	1	11
% Adeq. & Nonben.	5	3	8	2	7	16

Table 9.4: Comparision of policies to create cases in the domain of process planning.

	Dynamic					**Static**
Items	**1**	**2**	**3**	**4**	**5**	
% Cases Retr.	94	84	73	65	57	100
% Retr. Failures	37	27	15	9	6	41
% Nonben. Retr.	33	24	20	21	19	37
% Fail. & Ben.	18	10	5	2	1	24
% Adeq. & Nonben.	20	12	9	11	10	23

Table 9.5: Comparision of policies to create cases in the logistics transportation domain.

be succesful. This is particularly interesting in the logistics transportation domain where even though the percentage of retrieval decreases in a significant way, the percentage of nonbeneficial retrievals oscilates. Second, by observing the results of the whole collection (i.e., the 6th column), when no feature weights are considered, it can be observed that the concepts of retrieval failure is also independent of the fact that the retrieval is beneficial or not. We conclude that creating cases based on the benefits of the retrieval is a more adequate policy than creating cases based on retrieval failures. Two arguments can be given supporting this conclusion:

- If feature weighting are considered and the retrieval failures of a case are unlikely to occur, no new cases will be created. In the experiments, even though the difference between the case and the problem were at most of one goal, nonbeneficial retrievals were still likely to occur, particularly in the logistics transportation domain. If the difference is greater, say, a case achieves 4 goals and a new problem has 10 goals, it is clear that the retrieval is likely to be nonbeneficial.

- If feature weights are not considered, the independence between retrieval failures and benefits of a retrieval episode indicates that with the policy to create cases based on retrieval failures:

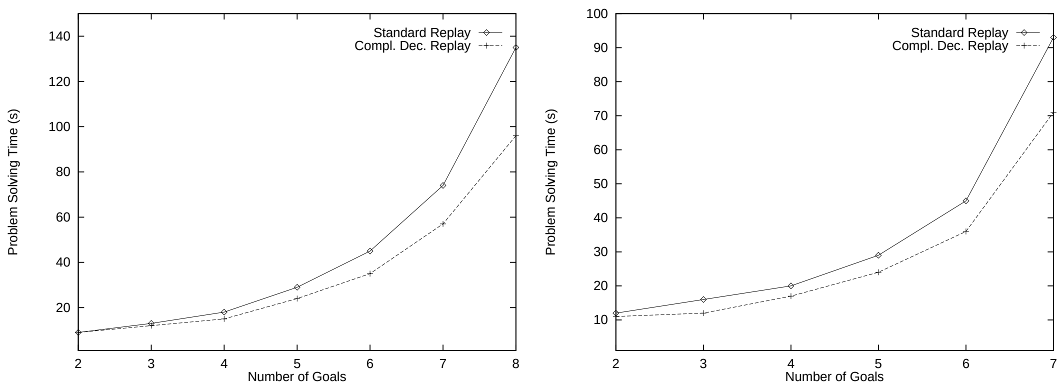


Figure 9.3: Problem solving times with standard and complete decision replay for (a) the domain of process planning and (b) the artificial domain.

- Cases may be created which can be generated with the existing cases with little effort.
- Situations may occur in which the adaptation is large but no new cases are created because no retrieval failure occurs.

9.4 Complete Decision Replay

To evaluate the performance of complete decision replay we used the same experiment setup as in Section 9.2 in which the performance of dependency-driven retrieval was evaluated for the domain of process planning. To isolate the effects of complete decision replay, we disable the feature weighting process and do not performed dependency-driven retrieval. This experiment was performed parallel to the one presented in Section 9.2. The cases were retrieved with the static mode (i.e., by using the architecture of the case base of Prodigy/Analogy). The retrieved case was adapted with standard replay and with complete decision replay.

Discussion of the Experiment Results. Figure 9.3 shows the results of the overall case-based, problem-solving process with standard and complete decision replay. As we see the performance is improved with complete decision replay; in the the domain of process planning and in the 7 in the artificial domain, the increase of performance was at least 20% (measuring the difference between the curves). As discussed in Section 7.3, a further performance improvement can be made if the retrieval failures can be predicted. In the same experiment we computed the overhead caused by the justification reconstruction process in the case-based, problem-solving episodes in which the retrieval was sucessful. The results were the following:

- 53% of the retrieval episodes were successful in the domain of process planning and 47% in the artificial domain.

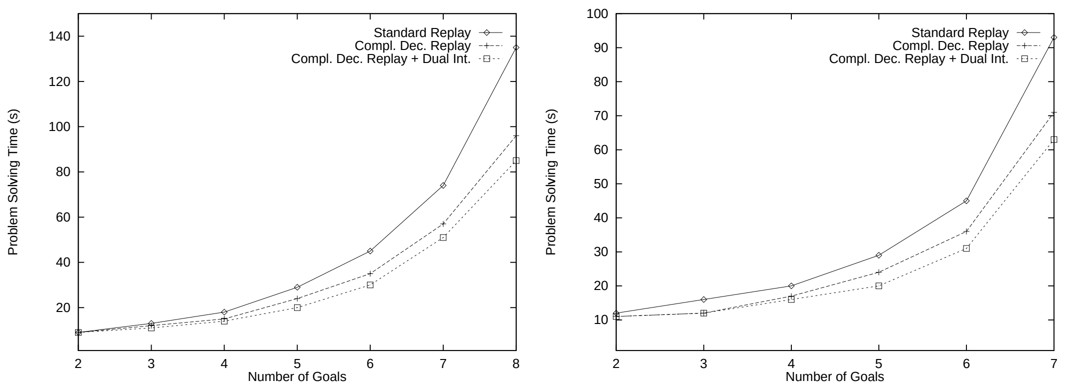


Figure 9.4: Problem solving times complete decision replay and the dual integration of retrieval and adaptation for (a) the domain of process planning and (b) the artificial domain.

- In these episodes, the overhead to the overall case-based, problem-solving process caused by the justifications reconstruction process was in average 39% in the domain of process planning and 34% in the artificial domain.

Experiment with the Dual Intergation between Retrieval and Adaptation. We repeated the previous experiment but this time we used the dual integration between retrieval and adaptation. The beneficial threshold was set to 2 to 1. Although no feature weighting process was performed, we kept track of the number of adequate and failed retrieval episodes, k_C and f_C, for each case C. The results are shown in Figure 9.4. The increase in performance relative to complete decision replay is small. However, compared to standard replay, the increase in performance was at least 25% for both domains in the 8-goal (7-goal) problems. Clearly, a major improvement should be expected if feature weighting is performed because this technique tends to improve the reliability of the retrieval. Thus, a more accurate prediction can be made.

9.5 Complete System

In the previous sections we evaluated the performance of each of the methods implemented in this thesis separately. In this experiment we wanted to show that the integration of the different techniques in CAPlan/CbC results in an effective problem-solver. As discussed in Chapter 7, this integration involves several issues including:

- The twofold retrieval process: combination of dependency-driven retrieval and feature weighting.

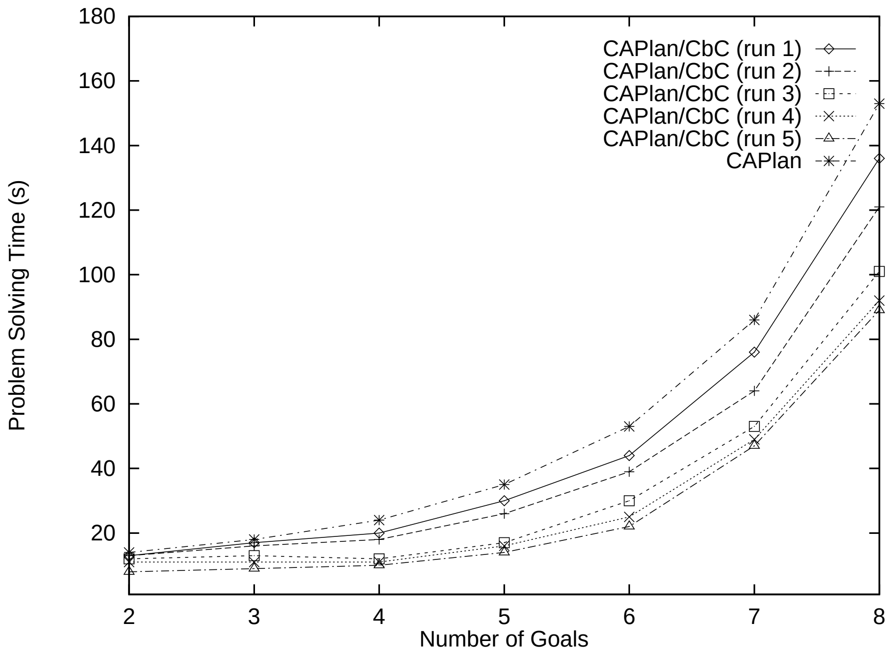


Figure 9.5: Performance gains with the complete case-based planner CAPlan/CbC over the base-level planner CAPlan.

- The dual integration of retrieval and adaptation: selection of the retrieved case and the adaptation method in the retrieval phase.
- The policy to create cases: creation of new cases if the retrieval is nonbeneficial.

The Experiment Setup. The experiment setup was designed in five runs. At each run a collection of problems was randomly formed. Each collection consisted of 2 one-goal problems, 4 two-goal problems, 8 three-goal problems, 10 four-goal problems and so on until 10 eight-goal problems. No two problems were the same within a collection or in different collections. This experiment setup allowed a better observation of the effects of the learning process. In addition, it was intended to be a fair simulation of realistic situations; by not saturating the case base first with 1-goal problems, then with 2-goal problems and so on, problem-solving episodes are likely to occur in which, say, to solve an 8-goal problem only a 2-goal case is available. This is particularly significant given our previous experiments that showed the independence between adequate retrieval and beneficial retrieval. The case base was maintained through all runs. The retrieval threshold was set to 75%. The beneficial threshold was set to 4 to 1 and the adaptation threshold was set to 2 to 1.

The Problem Domain. The problem domain was the domain of process planning. The ordering restrictions computed by the geometrical reasoner were given as part of the problem. A subset of nine types of processing was randomly selected from which the goals of the problems were selected.

Results. We measured the time for solving each problem by CAPLAN and CAPLAN/CBC at each run. The results are depicted in Figure 9.5. The values for CAPLAN correspond to the average of the five runs. Each run is correspondly labeled in each curve.

Discussion of the Experiment Results. We observe that with each run the performance of CAPLAN/CBC increases, showing that the methods for CBP proposed in CAPLAN/CBC are effectively integrated and that it improves its performance as more case-based problem solving episodes takes place. Given that at the first two runs, few learning opportunities take place, we conclude that the improvement there is mainly due to the dependency-driven retrieval technique. This is corroborated by the fact that the values obtained are in the same range of those in Figure 9.1. For the same reason, the further improvements in the next runs must be due to the learning process. A significant increase in the performance takes place between runs 2 and 3 and then the increase rate is relatively low. This suggests that at a certain point no further efficiency gains will be made. Overall, the increase in performance of the case-based problem solving process is at least of 40% in the fifth run.

9.6 Mergeability of Plans

We performed experiments to evaluate the different merging strategies in partially ordered plans based on SNLP.

Problem Domains. We performed experiments with the domain of process planning, the artificial domain and the logistics transportation domain.

Experiment Setup. We constructed a sequence of single-goal problems (15 for the domain of process planning, 12 for the transportation domain and 8 for the artificial domain). To observe the way that the positive interactions affect the merging process, sequences of $n+1$-goal problems were constructed by adding a goal randomly selected from the sequence of single-goal problems to each problem of the sequence of n-goal problems. In this way, we ensured that when several cases were retrieved, all the goals in the problem are covered. The constructed problems were revised to avoid repetitions. The construction of the case base reflects an ideal situation because in practice it is unlikely that all the goals of the problem are covered. However, this situation is appropiate to compare the merging methods because a maximal gain is expected from CBP. In this experiment, the feature weighting process was disabled, goal-driven retrieval and standard replay were performed. In this way we isolated the effects of the merging methods. The retrieved cases were merged in blind-merging and non-redundant-merging modes. For the domain of process planning the goals were merged in an order consistent with $\prec$ (although no dependency-driven retrieval was performed). The data obtained with SNLP (i.e., CAPLAN) is intended as a reference to compare the merging algorithms and not to show the advantages of CBP over first-principles planning.

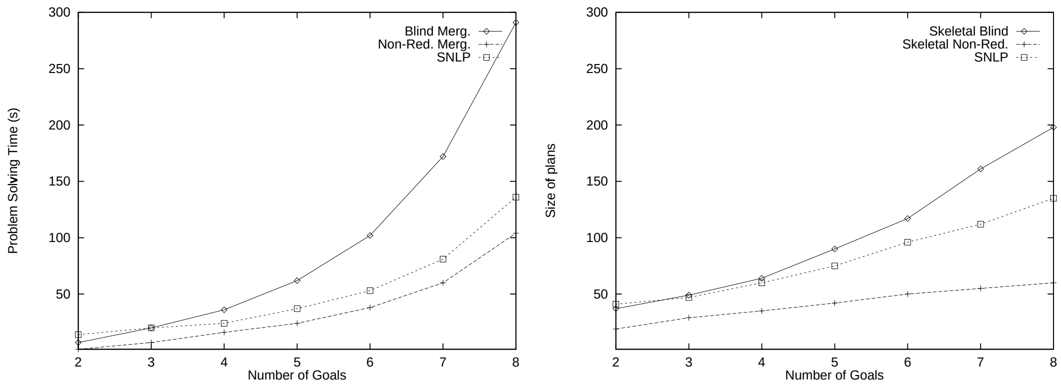


Figure 9.6: (a) Problem solving time and (b) plan size for the domain of process planning.

Results. Figures 9.6, 9.7 and 9.8 compare data obtained with the domain of process planning, the artificial domain and the logistics transportation domain. Part (a) of these figures shows the problem solving time by using blind and non-redundant merging and SNLP. Parts (b) shows the size (i.e., number of steps + number of ordering constraints in the plan) of the skeletal plans obtained with blind and non-redundant merging and the size of the solution plans obtained by SNLP.

Common to the three domains is that the worst performance is obtained with the blind merging mode (it is outperformed by SNLP). Blind merging seems to be

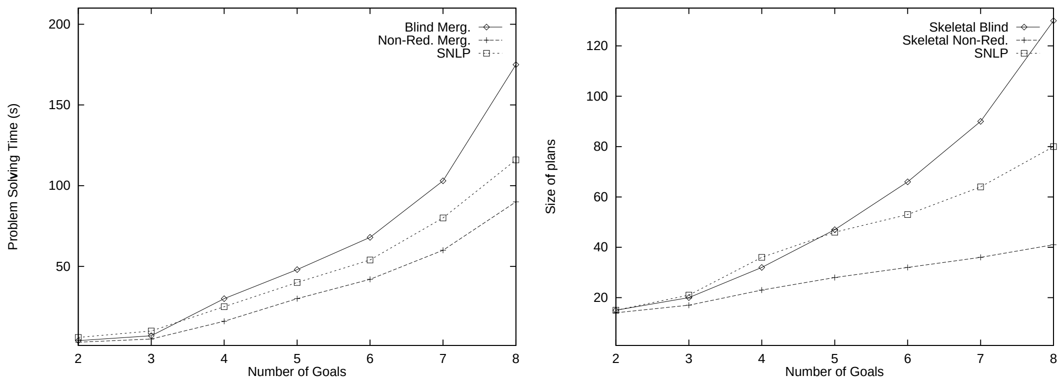


Figure 9.7: (a) Problem solving time and (b) size for the artificial domain.

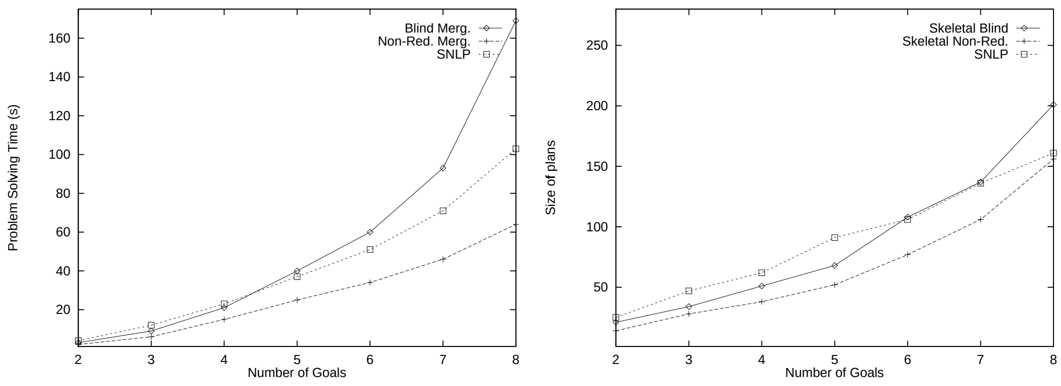


Figure 9.8: (a) Problem solving time and (b) skeletal plan size for the logistics transportation domain.

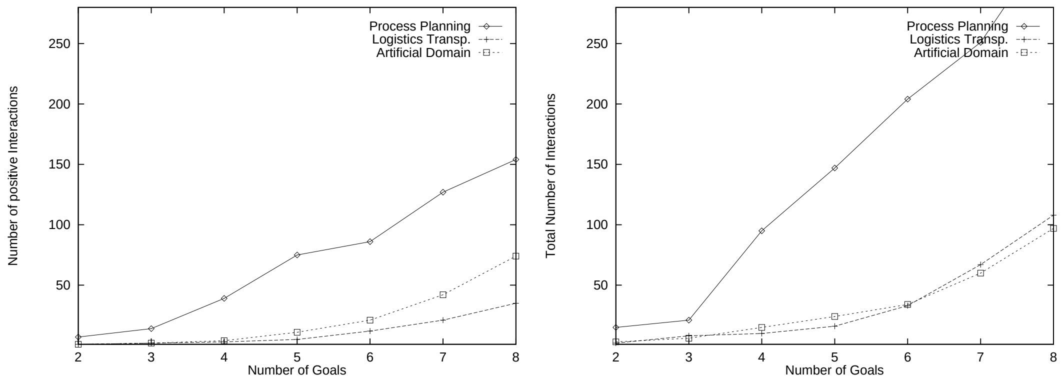


Figure 9.9: (a) Number of positive interactions and (b) total number of interactions.

an inadequate choice when the base level planner is SNLP. This result corroborates Proposition 8.2. In the domain of process planning and the artificial domain the result reflect the fact that subplans are always nonmergeable. In the transportation domain some of the skeletal subplans generated were mergeable. Thus, the performance with the blind merging mode is slightly better compared to the other domains. With the non-redundant mode, the difference between the size of the skeletal plan and the solution plan is a measure for the effort needed in the completion phase. For example, the effort for completition in the transportation domain (see Figure 9.8 (a)) is less than in the other two domains because the skeletal plans are comparatively larger (see Figure 9.8 (b)).

Figure 9.9 compares (a) the number of positive interactions and (b) the total number of interactions occuring in these domains. Because no positive interactions can occur after non redundant merging, the positive interactions are measured by observing the blind-merging mode. The difference between the skeletal plan in blind-merging mode with the skeletal plan in non-redundant-merging mode reflects the

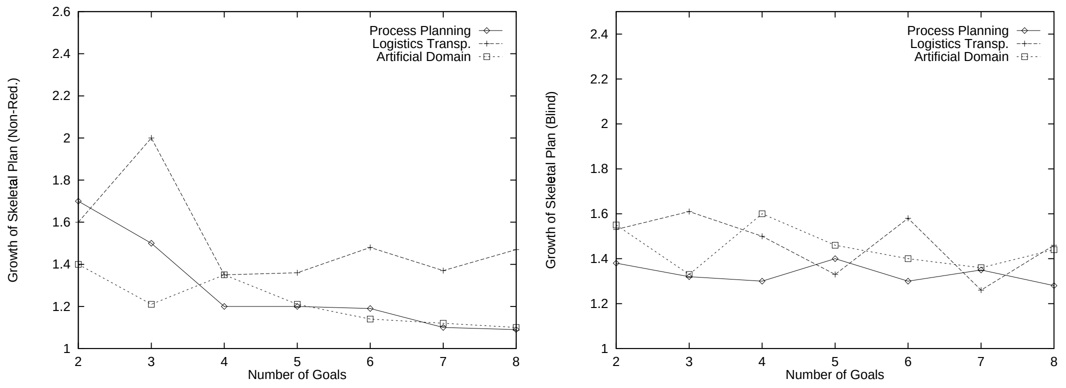


Figure 9.10: Growth of skeletal plan size with the (a) non-redundant and (b) blind merging.

percentage of the cases that was not replayed because of the positive interactions. Particularly, in the domain of process planning and the artificial domain, it can be observed that for solving the 8-goal problems, a significant part of the cases was not replayed (more than 70%). This is the result of the positive interactions, which in both situations correspond to approximately 50% of the interactions (see Figure 9.9).

Related to this issue, notice that as the number of goals increases, the guidance provided by the additional cases retrieved with the non-redundant merging mode decreases in the domain of process planning and the artificial domain. Figure 9.10 (a) compares the relative growth of the plans in the three domains for non-redundant merging whereas part (b) compares this growth for blind merging. The relative growth for n goals was measured by dividing the average size of the plans for n goals by the average size of the plans for n-1 goals. It can be observed that for the process planning domain and the artificial domain the relative growth of the skeletal plan generated in non-redundant mode decreases with the number of goals (see Figure 9.10). Notice that for both domains there is a significant increase in the number of positive interactions (see Figure 9.9). This supports our claim that in domains where positive interactions are likely to occur, the guidance provided by the additional cases retrieved tends to decrease with the number of goals. These results suggest that in these domains retrieving a single case covering as much of the goals as possible or fewer cases is an equivalent strategy because merging additional cases is worthless after several goals have been solved. Notice, that the growth of the skeletal plans obtained with blind merging does not decrease with the number of goals.

In contrast to the other two domains, in the logistics transportation domain the guidance provided by the additional cases with non redundant merging does not decrease as more goals have been solved and correspondingly the number of positive interactions does not increase in a significant way. This shows that in domains where goals are not necessarily in conflict, retrieving several cases and using non-redundant merging is indeed an adequate choice.

Part III

Conclusions and Related Work

Chapter 10

Related Work

In this chapter we will discuss work related to the thesis. We will first compare the problem-solving cycle in CAPLAN/CBC to a general problem-solving cycle in CBR. Next, we will compare methods for CBP implemented in other case-based planners with the methods implemented CAPLAN/CBC. Finally, relevant work in the field of machine learning will be discussed.

10.1 Case-Based Reasoning

CBR has been the subject of increasing interest in the AI community over a variety of fields ranging from academic research (Kolodner, 1993; Leake, 1996) to real-world applications (Althoff et al., 1995). Several problem-solving cycles have been proposed; one of which, presented in (Aamodt and Plaza, 1994), has been frequently referred. The cycle presupposes two knowledge sources; namely, the case base and general knowledge.[1] The cycle consists of four phases (see Figure 10.1):

Retrieval. Given a problem, one or more cases are selected which are rated as similar to the problem by a similarity assessment.

Reuse. The retrieved cases are used to find a solution of the new problem.

Revise. The obtained solution is examined to state if it is a valid solution of the problem. In this phase, the quality of the solution is also examined.

Retain. The revised solution is stored as a new case and the indexing structures are updated.

The problem-solving solving cycle in CAPLAN/CBC keeps the retrieval phase, the reuse phase, called adaptation, and the retain phase, called learning (see Figure

[1]A more concrete description of the knowledge in CBR has been given in (Richter, 1995); the knowledge of a CBR system is considered to be in four knowledge containers: the representation language, the similarity assessment, the adaptation mechanism, and the cases.

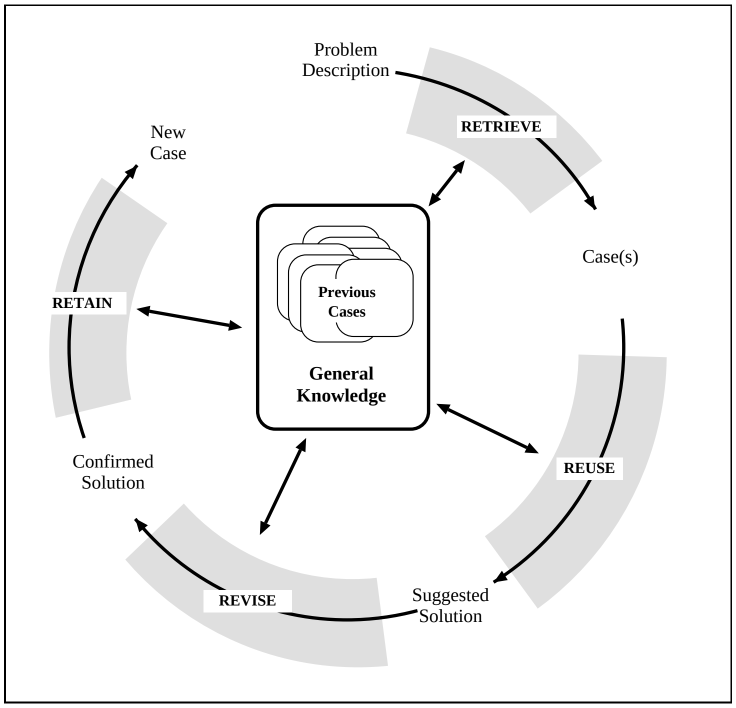


Figure 10.1: Problem solving cycle in CBR (adapted from (Aamodt and Plaza, 1994)).

3.1). The name adaptation was chosen as it is more specific; because the cases contain solution plans, reuse is made by adapting them. This is not the general situation in CBR. In classification tasks, for example, the term reuse is more appropriate as, in principle, the new problem is classified as the classification of the retrieved case. The name learning was chosen to emphasize that not only the obtained solution plans are possibly used to create a new case but that the system pursues to learn the best distribution of the feature weights by considering the feedback of the adaptation process.

There are three major differences between the problem-solving cycle in CAPlan/CbC and the one proposed in (Aamodt and Plaza, 1994):

The Revise Phase does not Takes Place. The solution obtained after the adaptation process in CAPlan/CbC is always correct. Thus, there is no necessity to revise the solution obtained. The reason for the correctness is based on the correctness of SNLP and the Justification-Reconstruction Claim (see Claim 5.1): first, SNLP does not make any restriction on which of the decisions of the conflict sets should be chosen. Thus, the fact that the decision chosen is the one indicated by the derivational path of the retrieved case is from the point of view of the correctness as good as another choice made by any other means. The key point is that the chosen decision belongs to the conflict set, which is precisely what the replay algorithm does (see step 1.4 of the algorithm *ReplaySubgoalGraph* depicted in Figure 5.4). Second, the Justifications Reconstruction Claim ensures that the decisions eliminated from the conflict sets during the justification reconstruction process are indeed invalid. Thus, if any of these decisions would have been chosen, no solution would have been found and SNLP would needed to backtrack on the choice made.

A New Phase: The Analysis Phase. This phase, which occurs previous to the retrieval process, receives as input the problem description and returns an extended problem description (see Section 3.2.1). This process could have been seen as a preprocessing to the retrieval phase. However, because none of the elements typical to the retrieval phase such as the case base or the similarity assessment are used to obtain the extended problem descriptions, we decided that this process should be distinguished from retrieval. In fact, the ordering constraints can be used to guide the planning process without using any CBR technique (Weberskirch, 1995).

Dual Integration Between the Retrieval and the Adaptation Phase. As briefly mentioned before, the output of the retrieval phase in the model of Aamodt and Plaza is one or more cases to be adapted. In CAPLAN/CBC not only the cases are selected but the adaptation method itself is selected (see Section 7.3). Even though the selection of the adaptation method is made based on the retrieved cases, the point is that this selection can only be made after retrieval had taken place.

We ommit further discussions about CBR in general. Extensive overviews can be found in (Kolodner, 1993; Leake, 1996).

10.2 Case-Based Planning

Several general-purpose case-based planners have been developed including:

Prodigy/Analogy. A pioneer work in the field (Veloso and Carbonell, 1993; Veloso, 1994); it implements for the first time a complete general-purpose, case-based, problem-solving cycle. Novel features include adaptation with analogical replay, the concept of relevant features and a complete architecture of the case base (some of these aspects were discussed in Section 2.6). PRODIGY/ANALOGY is based on the state-space planner Prodigy (Blythe et al., 1992).

Priar. Another early work in the field (Kambhampati, 1994); some of the novel features in Priar are cases representing a hierarchical plan, adaptation of hierarchical plans and rating of the features by their contribution to the cases.

SPA/MPA. SPA is a case-based planner performing single-case adaptation (Hanks and Weld, 1995) and MPA is the extension of SPA performing multiple-case adaptation (Francis and Ram, 1995b). The most important contribution of this work is an adaptation algorithm in which the solution of the case is *transformed* into a solution of the new problem. This is opposed to adaptation based on replay in which the derivational trace is reconstructed relative to the new solution. Unfortunately, even though SPA has been shown to outperform first-principles planning with SNLP, no report has been made comparing it with adaptation based on replay.

derSNLP+EBL. The case-based planner derSNLP+EBL performs case adaptation with standard replay based on SNLP (Ihrig and Kambhampati, 1996a). If a retrieval failure occurs (see Definition 6.3), EBL generates a rule explaining the failure. Each of the retrieved cases is annotated with the rule. The censoring rule serves as a choice node in the case base between each of them and a new case containing the solution obtained when the cases were retrieved and the failure occured. In subsequent retrieval episodes, before selecting any of the censored cases, the retrieval procedure checks that the rule does not hold. Otherwise, the new case is retrieved. The new case may be censored as well, if it is retrieved and a retrieval failure occurs.

MRL. The inference mechanism of MRL is based on deductive planning (Koehler, 1994). The deduction mechanism is used to generate plans. The functionality obtained is somehow comparable to state-space as the state of the world is represented and transformed in the deductive formalism.

Paris. Paris reuses abstract cases (Bergmann and Wilke, 1995b). Given a new problem, its problem description is abstracted. The case base is searched for a case solving the abstracted problem description. Once such a case is found its solution is refined to a concrete level. New cases are created by abstracting the solutions found. Paris is based on a linear, state-space planner.

Table 10.1 compares these systems with CAPLAN/CBC according to the following aspects (see Chapter 1):

1. **Goal-driven retrieval, dependency-driven retrieval or other forms of retrieval**. Goal-driven retrieval means that the first aspect of the problem descriptions considered during retrieval are the goals to be achieved. Dependency-driven retrieval means that the first aspect considered are the ordering restrictions of the problem. Dependency-driven retrieval presupposes that ordering restrictions can be precomputed (see Sections 3.2.1 and 2.5). Priar and Paris presuppose in addition to the availability of the domain theory (a common requirement for any general-purpose planner), a transformation theory between hierarchies and abstractions levels respectively. MRL requires a formalization in terminological logic for indexing the cases.

2. **Knowledge Necessary for the Problem Solving Cycle.** As mentioned in 1, all case-based planners require at least the domain theory. Some require additional knowledge. For example, in CAPLAN/CBC, ordering restrictions are required to perform dependency-driven retrieval. However, in some domains, the ordering restrictions can be generated automatically from the domain theory (see Section 2.5). Thus, for those domains, no additional knowledge is necessary.

3. **Static or dynamic similarity metrics**. Static means that the value of the similarity assessment between a problem and a case is always the same. Dynamic means that this value may change after case-based problem solving episodes have taken place. The fact that the similarity metric in a system is static does not means that given a certain problem, the same case will always be retrieved because new cases might have been created. The point here is that the similarity metric learns from previous case-based problem solving episodes.

4. **Base-level planner searches in the space of states or in the space of plans**. Adaptation based on replay is not the only adaptation method that leaves part of the refitting effort to the base-level planner. In Paris, for example, first-principles planning is done to refine the abstract cases to a solution plan. Thus, the search space of the base level planner is also an important factor to be considered in CBP.

5. **Adaptation based on standard replay, adaptation based on replay but considering failed attempts or other forms of adaptation**. The difference between the first two is that in the former only the derivational path driving to the solution is considered during the adaptation process whereas, in the latter, in addition to this path, the decisions taken when the cases were solved and which drove to no solution are also considered. The third possibility refers to adaptation methods which are not based on replay. PRODIGY/ANALOGY and CAPLAN/CBC take into account the failed attempts although the former searches in the space of states and the latter in the space of plans. Thus, the mechanisms to represent the failed attempts and to handle this information are completely different.

6. **The system supports user interactions during the adaptation process**. Remarkably, no general-purpose, case-based planner has considered this issue before.

7. **Eager policy to create new cases, case creation based on retrieval failures or case creation based on retrieval benefits**. The first means that every time a new solution is found, new cases are created. The second one means that a new case is created if a retrieval failure occurs and the last one means that a new case is created if the retrieval is nonbeneficial.

Issue	Prodigy/ Analogy	Priar	SPA/ MPA	derSNLP +EBL	Paris	MRL	CAPlan/ CbC
Retrieval	goal	hierar.	goal	goal	abstr.	other	*depend.*
Addition. Know-ledge	none	hierar. theory	none	none	abstr. theory	termin. logic	*none or goal orderings*
Simil. Metric	static	static	static	static	static	static	*dynamic*
Search Space	states	plans	plans	plans	states	states	*plans*
Adap-tation	replay +just.	hierar. adap.	transf.	replay	refinement	other	*replay +just.*
User Inter.	no	no	no	no	no	no	*yes*
Case Creation	eager	eager	eager	failure	eager	eager	*benefit*
Archi-tecture	yes	no	yes	yes	yes	yes	*yes*
Feedback	yes	no	no	yes	yes	no	*yes*
One or Multi-Case Retriev.	multi	one	multi	multi	one	one	*multi*

Table 10.1: Comparison between several general-purpose, case-based planners on selected issues.

8. **Architecture of the case base**. Providing an indexing structure is key to evaluate the similarity assessment efficiently. For this reason, most of the systems provide an architecture of the case base.

9. **Feedback from the problem solver**. The feedback is used to improve the accuracy of the retrieval. CBR+EBL (Cain et al., 1991) is a domain-specific case-based reasoner, which used EBL to improve the accuracy of the retrieval.[2] In PRODIGY/ANALOGY, the feedback is used to rank the features in the case base; features with a higher rank are to be matched before features with lower rank. The rank of the features, however, is not consider to evaluate the similarity. That is, the rank of the feature is not a weight and thus is not taken into account in the similarity assesssment. derSNLP+EBL uses feedback of the problem solver to censor the retrieval of a case by adding EBL rules every time a retrieval failure occurs. Thus, to the usual retrieval costs, the cost of evaluating the rules should be added. This may result in an architectual utility problem (Francis and Ram, 1993), an extension of Minton's utility problem (Minton, 1988). This problem arises when learning has the side effect of causing an increase in the costs of the basic operations that the system performs. The basic operation in this situation is retrieval and the increased in costs are caused by the evaluation of the EBL rules. In our approach, the feedback results in a redistribution of the feature weights but no new elements need to be evaluated.

10. **Single or Multi-Case Retrieval.** The ability to merge multiple cases is another factor that distinguishes different case-based planners. However, the results of Chapter 8 shows that there are some limitations for case-based planners based on SNLP.

10.3 Mergeability of Plans

The idea of avoiding redundancy during replay of multiple cases was first proposed in Prodigy/Analogy. Prodigy/Analogy uses a mixed-initiative strategy to switch the search control between case-based and first-principles. The basis for this strategy is the fact that Prodigy searches in the space of states instead of the space of plans as in CAPLAN/CBC. Selecting the kind of base-level planner depends on the characteristics of the particular domain (Kambhampati et al., 1996a; Barrett and Weld, 1994; Veloso and Blythe, 1994). For example, our specification of the domain of process planning (see Appendix A), there is theoretical evidence that a partial-order planner such as SNLP is a better choice (Muñoz-Avila and Weberskirch, 1996c).

derSNLP+EBL applies the following strategy: in principle, several single-goal cases are retrieved, each covering a goal of the problem. Non-redundant merging

[2]CBR+EBL is a domain-specific system and as such it is not compare in Table 10.1.

is used to combine them. If decisions in the subplans achieving the goals need to be revised to obtain a solution, this solution is stored as a new multi-goal case. If in future retrieval episodes the same situation is encountered, the multi-goal case is retrieved instead of the single-goal cases. Based on the results of this paper, we affirm that in domains where goals are in conflict this method results in an improvement of the performance of the planning process as the result of the multi-goal cases learned but not of the merging method itself. As the number of goals increases, the process of merging cases serves mainly to construct multi-goal cases.

Although not in the context of partial-order planning, previous work has shown that merging subplans into a solution is NP-complete (Karinthi et al., 1992; Yang et al., 1992). The same work, however, shows that there are instances of the problem that can be solved in polynomial time. An algorithm for merging is presented containing several operations, one of which involves merging the same step occuring in different plans into a single step. This operation is comparable to prefering existing establishing oportunities of the non redundant merging method.

10.4 Machine Learning

Feature weighting has been the subject of continuous research in machine learning (e.g., (Aha and Goldstone, 1990; Salzberg, 1991; Skalak, 1992; Wettschereck and Aha, 1995)). Moreover, the meaning of the context of a feature has been studied as well. For example, in (Aha and Goldstone, 1990), it has been pointed out that the relevance of a feature is a context-specific property. Typically, the notion of relevance and context of a feature has been defined in terms of statistical information such as the distribution of the feature values relative to the values of other features (e.g., in (Turney, 1996)). An overview of feature weighting can be found in (Atkenson et al., 1995).

A common characteristic of previous work on feature weighting is that it has been done for analysis tasks such as classification, but not for synthesis tasks such as planning. There are two key differences between these tasks which determine the way the relevance and the context of a feature are handled in CBP:

Classification problems have a single solution whereas planning problems typically have several solutions. This is an important factor to determine the relevance of a feature in CBP; Veloso pointed out that the relevance of a feature depends on the particular solution plan found (Veloso, 1994). The foot-printing process determines the relevant features of a particular solution.

Typically, no domain theory is available in classification whereas the domain theory is always available in general-purpose CBP. As described in Section 6.3, the domain theory plays a key role in determining the context of a feature in CBP; if goals in a domain are known to be ($\prec$-constrained) trivially serializable, goals that occur in the new problem but not in the retrieved case but are not part of the context of the case features.

As mentioned in Section 6.1, the feature weighting model in CAPLAN/CBC is based on incremental optimizers (Wettschereck and Aha, 1995). More concrete, our model is an extension of Salzbergs model (Salzberg, 1991), which can be stated as follows:

$$\omega_{i,C} = \begin{cases} \omega_{i,C} + \triangle & : \textit{correct retrieval} \\ \omega_{i,C} - \triangle & : \textit{false retrieval} \end{cases}$$

There are two main differences to our model. First, because Salzbergs model was developed for classification tasks, the weights of the features are increased (decreased) if the retrieval is correct (false). That is, if the classification of the retrieved case is the classification of the problem. Clearly, in CBP, it makes no sense to call a retrieval correct (or false) because the case can always be adapted. Whether the retrieval was effective or not is another issue. Thus, in our model we speak of an *adequate* retrieval (or a retrieval *failure*).

Second, the *incremental factor*, $\triangle$, is a global, constant factor in Salzbergs model. In contrast, in our model, the incremental optimizer, $\triangle_{k_C,f_C}$, is local and varies according to the number of adequate and inadequate retrievals of each case C (i.e., k_C, f_C). The reason for this difference are the purposes of each method; Salsbergs purpose is to predict the classification of points in the hyperspace (i.e., in $R^n \times R^n$). To achieve this, the hyperspace is divided in hyper-rectangles, representing generalizations of the points for which the clasification is known. The classification of a new point is predicted by voting on the classifications of the hyper-rectangles containing it.[3] In our approach, the purpose of the feature weighting process is to improve the accuracy of the retrieval of each case. That is, to increase the possibility that when a case is retrieved, the solution of the case can be extended to a solution plan of the new problem. Thus, we are interested on each case locally. As such, the incremental factor of a case is independent of the other cases. Our approach is comparable to research done on local similarity metrics (e.g., (Ricci and Avesani, 1995)) because the weighted similarity metric can be seen as a collection of local similarity metrics; one for each case.

[3]In (Wettschereck and Diettrich, 1994), some flaws of Salzbergs approach have been pointed out; the generalizations obtained with the hyperrectangles turn out to be over-generalizations. Further, Salzbergs model is outperformed by classification methods based on nearest neighbour.

Chapter 11

Conclusions and Future Research Directions

We conclude this thesis by summarizing the obtained results and discussing future research directions.

11.1 Summary and Conclusions

In this thesis we presented novel methods for CBP, which were motivated by problems encountered in complex domains such as the domain of process planning. These methods are:

Dependency-Driven Retrieval. In domains in which valid ordering restrictions can be stated previous to the problem-solving process, dependency-driven retrieval improves the performance and the accuracy of the retrieval process compared to goal-driven retrieval. The GDN is an effective indexing structure by enabling CAPLAN/CBC to evaluate the order consistency condition on several cases at the same time. However, in domains in which no valid ordering restrictions can be stated, the GDN is not an effective indexing structure because retrieval costs increase in a significant way.

Dynamic Similarity Metrics. A dynamic similarity metric, the weighted similarity metric, has been stated. This metric assesses similarity by counting the weights of the features in the candidate cases matching features of the new problem. The feature weights of a case are updated by considering the feedback of the adaptation process; if the retrieval is adequate, the weights of certain features is incremented by a factor which depends on previous performances of the case. Otherwise, the weights of the features are decremented by the same factor. The features whose weight is updated are identified by a process called filtering, in which the contributions of the features to the skeletal plan are analyzed.

The Context of a Feature. The purpose of updating the feature weights is to rank the case features according to their relevance. The process presupposes that the context of the case features are the features of the problem and the case and the common goals in the problem and the case. However, we saw that there are situations in which the failure of the retrieval is due to goals in the new problem which are conflicting with the goals of the case. In these situations, the absence of case features in the new problem has no effect whatsoever on the outcome of the retrieval. Thus, the feature weights should not be updated. We found that in these situations, goals are not trivially serializable. We concluded that in domains for which goals are known to be trivially serializable, feature weighting can be performed without encountering such situations. In the general case, that is, in domains for which goals may not be trivially serializable, EBL can be used to detect these situations. We also found that the requirement of trivial serializability between the goals can be weakened if valid ordering restrictions, $\prec$, between the goals can be predefined. In this situation, the requirement to simplify the context is that goals are $\prec$-constrained trivially serializable.

Adaptation in Plan-Space Planning with Complete Decision Replay. The base-level planner CAPLAN maintains the subgoal graph, a structure representing dependencies between planning elements such as goals and decisions. In addition, the justifications for every decision are constructed. CAPLAN/CBC stores as part of the cases the subgoal graph and the justifications. Complete decision replay reconstruct the subgoal graph relative to the new problem. This corresponds to standard replay. In addition, the justifications of the failed decisions (i.e., decisions that were taken during planning but were rejected because they did not conduct to any solution plan) are reconstructed relative to the new problem. If the justifications of a failed decisions of the case can be reconstructed, the decision also fails in the new problem. Thus, it can be discarded. As a result, the search space that the base-level planner needs to explore to complete the skeletal plan can be smaller compared to standard replay. Two additional advantages are the result of inheriting the functionality of CAPLAN and the fact that CAPLAN/CBC reconstructs the subgoal graph and the justifications. First, the user may interact with the system during the adaptation process and, second, powerful backtracking mechanism such as dependency-directed backtracking can be performed during the adaptation process.

Trade-off between Efficiency Gains and Case Merging. Problem solving by blind merging the retrieved cases is known to result in solution plans containing redundant steps. We found that when the base-level planner is SNLP, not only the resulting plans contain redundant steps but that the planning process itself is very inefficient. We found that a reason for this is that, for SNLP, the concept of mergeability is equivalent to the concept of trivial mergeability. Thus, unless the retrieved cases can be extended to a solution plan by adding only ordering and binding constraints, backtracking will always take place.

Previous work has shown that if there is a point in which it does not payoff to spend further effort in the retrieval phase because no gains will be made in the overall case-based, planning process. We found a somewhat similar result for adaptation with non-redundant merging; namely, in domains in which positive interactions frequently occur, the increase in size of the skeletal plan reduces with the number of goals and the number of cases that is been merged. Thus, there is a point in which it does not payoff to merge further cases.

Policy to Create New Cases based on Retrieval Benefits. Previous work created cases either eagerly (i.e., every time a new solution is found) or if a retrieval failure occurs. A flaw of the eager policy to create cases is that case bases tend to be very large. This is partially corrected by the policy based on retrieval failures. However, we found that this policy has two flaws: first, situations may occur in which no retrieval failure occur but still the adaptation effort was large. Second, there are situations in which a retrieval failure occurs, but the adaptation effort is small. To solve these flaws we state a policy to create cases based on the retrieval benefits; new cases are created only when the adaptation effort is large. The measure of the adaptation effort is made with an heuristic assessment.

Experiments were performed to further illustrate the effectiveness of these methods individualy. However, this thesis is not a collection of independent methods improving different aspects of CBP, but it integrates these methods in the problem solving cycle of CAPlan/CbC. The purpose of this integration is to compensate the weak points of some of these methods with the strengths of the others. This integration involves the following aspects:

Twofold Retrieval. The first integration aspect is the twofold retrieval process. Dependency-driven retrieval and feature weighting are complementary steps of the retrieval process because the first performs a preselection based on the ordering restrictions and the goals and the second makes a final selection based on the initial features. Moreover, the strenght of feature weighting is that it decreases the possibility that retrieval failures occur by learning from previous retrieval episodes. Its main drawback is that, under circumstances, several retrieval episodes may take place until a good distribution of weights is learned. Dependency-driven retrieval is, on the contrary, static because it does not learn from previous retrieval episodes. However, dependency-driven retrieval is a rather reliable process.

Dual Integration of Retrieval and Adaptation. The second integration aspect is the integration between feature weighting and complete decision replay. As discussed before, the feature weighting process decreases the possibility that retrieval failures occur but, under circumstances, it may take several retrieval episodes before a good distribution of weights is learned. If the retrieval of a case is known statistically to be unreliable, CAPLAN/CBC performs complete decision replay. The reason for this, is that when retrieval failures occur, backtracking must take place. By performing complete decision replay, invalid alternatives to complete the skeletal plan are discarded and, thus, the completion effort is reduced. A drawback of complete decision replay is the overhead cased by the justification process. If after several retrieval episodes, the possibility of retrieval failures decreases (as it is the usual result of the feature wieghting process), CAPLAN/CBC performs standard replay because backtracking on the skeletal plan is unlikely to occur.

In this thesis, several contributions to each of the phases of the case-based planning process have been made. The weakeness of some of the proposed methods was compensated with the advantages of others. The result is a powerful case-based planner, CAPLAN/CBC, which advances the state of the art in CBP.

11.2 Future Research Directions

There are three main research directions that we believed should be explored. First, given the advances of CBP, an integration of different CBP techniques can be made. UCP is a planning system integrating different planning paradigms such as state-space and plan-space planning (Kambhampati, 1996). Thus, it may serve as a base for exploring this integration. However, in UCP no learning can be made on the selection of the planning paradigm; if for example, in a certain point of the planning process, the system switches from state-space search mode to plan-space search mode, there is no opportunity to learn from this decision. The reason for this is that UCP will always be able to find a solution in any planning mode. This is an interesting property as it ensures the completness of UCP. But, this also means that no learning can be made on the decision of which planning mode to use. Even though in CBP no learning is made on single decisions but in the whole planning process, this fact may pose problems for any learning method including CBP. The integration should not necessarily be done based on the different search spaces in which planning may take place; in some of the experiments performed in this thesis, the architecture of PRODIGY/ANALOGY was integrated into CAPLAN. This is particularly usefull if no ordering restrictions are given because in this situation the GDN will decrease the performance of the system. Following similar motivations, a systematic integration of the different CBP techniques can be explored.

The second issue is the application of CBP in the context of information gathering. Information gathering has been the subject of increasing interest because of its wide range of practical applications such as search in internet and information retrieval (Selberg and O.Etzioni, 1995; Pryor, 1995; Knoblock, 1996; Etzioni et al., 1996; Golden and Weld, 1996). CAPLAN/CBC as all other case-based planners assumes that all conditions relevant to solve the problem are known previous to the planning process. This assumption is somewhat relaxed in CAPLAN/CBC by considering interactions of the user. However, these interactions are limited to pruning parts of the plan and to make preconditions invalid. In information gathering, more functionality is needed; some conditions are known previously but others are only known during the planning process. Further, which conditions are known at a certain point of the planning process dependend on the partial solution being obtained.

The last research direction concerns the automatic computation of the ordering restrictions, which are necessary to perform dependency-driven retrieval. As discussed in this thesis, Etzionis STATIC can be used to compute automatically ordering restrictions. However, Etzioni's procedure presupposes that axioms are stated in addition to the domain theory. A more powerful procedure should be developed that states the ordering restrictions solely based on the domain theory.

Part IV

Appendix

Appendix A

The Domain of Process Planning

Object types: (33)

CenterOutline
Superclass: Outline

DrillTool
Superclass: HoleTool

ExtRotaryTool
Superclass: Tool

Feature
Superclass: ProcArea

FeatureTool
Superclass: Tool

Groove
Superclass: Feature

GrooveTool
Superclass: FeatureTool

Hole
Superclass: Feature

HoleTool
Superclass: FeatureTool

IntProcArea
Superclass: ProcArea

IntRotaryTool
Superclass: Tool

LeftOutline
Superclass: Outline

LeftRTool
Superclass: ExtRotaryTool

Outline
Superclass: ProcArea

PrickOut
Superclass: Feature

PrickOutTool
Superclass: FeatureTool

ProcArea
Superclass: None.
RightOutline
Superclass: Outline
RightRTool
Superclass: ExtRotaryTool
RoundOff
Superclass: Feature
RoundOffTool
Superclass: FeatureTool
Slope
Superclass: Feature
SlopeTool
Superclass: FeatureTool
TappingTool
Superclass: HoleTool
Thread
Superclass: Feature
ThreadTool
Superclass: FeatureTool
Tool
Superclass: None.
Undercut
Superclass: Feature
UndercutHalf1
Superclass: Undercut
UndercutHalf2
Superclass: Undercut
WpieceSide
Superclass: ProcArea
WpieceSide1
Superclass: WpieceSide
WpieceSide2
Superclass: WpieceSide

Predicates: (17)
available
Arguments: tl
clampNoTurn
Arguments: pos
clampTurn
Arguments: pos
isClampArea
Arguments: area
neighbour
Arguments: area1 area2
noSubareaThread
Arguments: outl

processed
Arguments: area

processedUcutHalf1
Arguments: ucut

processedUcutHalf2
Arguments: ucut

restrictedClamp
Arguments: area

subarea
Arguments: feat outl

toolHeld
Arguments: tl

toolHolderFree
Arguments: None.

unprocessed
Arguments: area

unprocUcutHalf1
Arguments: area

unprocUcutHalf2
Arguments: area

unrestrictedClamp
Arguments: outl

Operators: (27)

ClampFromFace

Arguments:	s1 s2 outl1 outl2 inn
Constraints:	IsOfType(Outline, outl1)
	IsOfType(Outline, outl2)
	NotSame(outl2, outl1)
	IsOfType(WpieceSide, s2)
	IsOfType(WpieceSide, s1)
	NotSame(s2, s1)
	IsOfType(IntProcArea, inn)
Effects:	+clampTurn(s1)
	-clampTurn(outl1)
	-clampNoTurn(outl1)
	-clampTurn(outl2)
	-clampNoTurn(outl2)
	-clampNoTurn(s2)
Preconditions:	+subarea(inn, s1)
	+processed(inn)
	+unprocessed(s2)
	+isClampArea(s1)

ClampNoTurn

Arguments:	outl s2 s outl2
Constraints:	IsOfType(Outline, outl2)
	IsOfType(Outline, outl)
	IsOfType(WpieceSide, s2)
	IsOfType(WpieceSide, s)
	NotSame(outl, outl2)
	NotSame(s2, s)
Effects:	+clampNoTurn(outl)
	-clampTurn(outl2)
	-clampNoTurn(outl2)
	-clampNoTurn(s)
	-clampNoTurn(s2)
	-clampTurn(outl)
Preconditions:	+noSubareaThread(outl)
	+isClampArea(outl2)
	+isClampArea(outl)
	+unrestrictedClamp(outl)

ClampNoTurn-NotFree

Arguments:	outl s2 s outl2
Constraints:	IsOfType(Outline, outl2)
	IsOfType(Outline, outl)
	IsOfType(WpieceSide, s2)
	IsOfType(WpieceSide, s)
	NotSame(outl, outl2)
	NotSame(s2, s)
Effects:	+clampNoTurn(outl)
	-clampTurn(outl2)
	-clampNoTurn(outl2)
	-clampNoTurn(s)
	-clampNoTurn(s2)
	-clampNoTurn(outl)
Preconditions:	+unprocessed(outl)
	+isClampArea(outl2)
	+isClampArea(outl)
	+restrictedClamp(outl)

ClampNoTurn-Thread

Arguments:	outl s2 s outl2 thr
Constraints:	IsOfType(Outline, outl2)
	IsOfType(Outline, outl)
	IsOfType(WpieceSide, s2)
	IsOfType(WpieceSide, s)
	NotSame(outl, outl2)
	NotSame(s2, s)
	IsOfType(Thread, thr)
Effects:	+clampNoTurn(outl)
	-clampTurn(outl2)
	-clampNoTurn(outl2)
	-clampNoTurn(s)
	-clampNoTurn(s2)
	-clampTurn(outl)
Preconditions:	+unprocessed(thr)
	+subarea(thr, outl)
	+isClampArea(outl2)
	+isClampArea(outl)

ClampTurn

Arguments:	outl s2 s outl2
Constraints:	IsOfType(Outline, outl2)
	IsOfType(Outline, outl)
	IsOfType(WpieceSide, s2)
	IsOfType(WpieceSide, s)
	NotSame(outl, outl2)
	NotSame(s2, s)
Effects:	+clampTurn(outl)
	-clampNoTurn(s)
	-clampNoTurn(s2)
	-clampTurn(outl2)
	-clampNoTurn(outl2)
	-clampNoTurn(outl)
Preconditions:	+noSubareaThread(outl)
	+isClampArea(outl2)
	+isClampArea(outl)
	+unrestrictedClamp(outl)

ClampTurn-NotFree

Arguments:	outl s2 s outl2
Constraints:	IsOfType(Outline, outl2)
	IsOfType(Outline, outl)
	IsOfType(WpieceSide, s2)
	IsOfType(WpieceSide, s)
	NotSame(outl, outl2)
	NotSame(s2, s)
Effects:	+clampTurn(outl)
	-clampTurn(outl2)
	-clampNoTurn(outl2)
	-clampNoTurn(s)
	-clampNoTurn(s2)
	-clampNoTurn(outl)
Preconditions:	+unprocessed(outl)
	+isClampArea(outl2)
	+isClampArea(outl)
	+restrictedClamp(outl)

ClampTurn-Thread

Arguments:	outl s2 s outl2 thr
Constraints:	IsOfType(Outline, outl2)
	IsOfType(Outline, outl)
	IsOfType(WpieceSide, s2)
	IsOfType(WpieceSide, s)
	NotSame(outl, outl2)
	NotSame(s2, s)
	IsOfType(Thread, thr)
Effects:	+clampTurn(outl)
	-clampTurn(outl2)
	-clampNoTurn(outl2)
	-clampNoTurn(s)
	-clampNoTurn(s2)
	-clampNoTurn(outl)
Preconditions:	+unprocessed(thr)
	+isClampArea(outl2)
	+isClampArea(outl)
	+subarea(thr, outl)

DrillHole

Arguments:	hole outl aTool clampArea
Constraints:	IsOfType(DrillTool, aTool)
	IsOfType(Outline, outl)
	IsOfType(ProcArea, clampArea)
	IsOfType(Hole, hole)
	NotSame(outl, clampArea)
Effects:	+processed(hole)
	-unprocessed(hole)
Preconditions:	+processed(outl)
	+clampNoTurn(clampArea)
	+toolHeld(aTool)
	+subarea(hole, outl)
	+available(aTool)

HoldTool

Arguments:	tool1
Constraints:	None.
Effects:	+toolHeld(tool1)
	-toolHolderFree()
Preconditions:	+toolHolderFree()

MachineGroove

Arguments:	groove outl tool clampArea
Constraints:	IsOfType(GrooveTool, tool)
	IsOfType(Outline, outl)
	IsOfType(ProcArea, clampArea)
	IsOfType(Groove, groove)
	NotSame(outl, clampArea)
Effects:	+processed(groove)
	-unprocessed(groove)
Preconditions:	+processed(outl)
	+clampTurn(clampArea)
	+toolHeld(tool)
	+subarea(groove, outl)
	+available(tool)

MachineInternalArea

Arguments:	inn s tool outl clampArea
Constraints:	IsOfType(IntProcArea, inn)
	IsOfType(WpieceSide, s)
	IsOfType(IntRotaryTool, tool)
	IsOfType(Outline, outl)
	IsOfType(Outline, clampArea)
	NotSame(outl, clampArea)
Effects:	+processed(inn)
	-unprocessed(inn)
Preconditions:	+available(tool)
	+processed(s)
	+clampNoTurn(clampArea)
	+toolHeld(tool)
	+subarea(inn, s)
	+neighbour(outl, s)

MachineOutline

Arguments:	outl tool clampArea
Constraints:	IsOfType(Outline, outl)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
	IsOfType(ExtRotaryTool, tool)
Effects:	+processed(outl)
	-unprocessed(outl)
Preconditions:	+clampTurn(clampArea)
	+toolHeld(tool)
	+unprocessed(outl)
	+available(tool)

MachinePrickOut

Arguments:	prickout outl tool clampArea
Constraints:	IsOfType(PrickOutTool, tool)
	IsOfType(Outline, outl)
	IsOfType(ProcArea, clampArea)
	IsOfType(PrickOut, prickout)
	NotSame(outl, clampArea)
Effects:	+processed(prickout)
	-unprocessed(prickout)
Preconditions:	+processed(outl)
	+clampTurn(clampArea)
	+toolHeld(tool)
	+subarea(prickout, outl)
	+available(tool)

MachineRoundOff

Arguments:	roundoff outl tool clampArea
Constraints:	IsOfType(RoundOffTool, tool)
	IsOfType(Outline, outl)
	IsOfType(ProcArea, clampArea)
	IsOfType(RoundOff, roundoff)
	NotSame(outl, clampArea)
Effects:	+processed(roundoff)
	-unprocessed(roundoff)
Preconditions:	+processed(outl)
	+clampTurn(clampArea)
	+toolHeld(tool)
	+subarea(roundoff, outl)
	+available(tool)

MachineSide

Arguments:	s clampArea tool s1
Constraints:	IsOfType(Outline, clampArea)
	IsOfType(WpieceSide, s)
	IsOfType(WpieceSide, s1)
	IsOfType(LeftRTool, tool)
	NotSame(s, s1)
Effects:	+processed(s)
	-unprocessed(s)
Preconditions:	+clampTurn(clampArea)
	+toolHeld(tool)
	+unprocessed(s)
	+neighbour(clampArea, s1)
	+available(tool)

MachineSlope

Arguments:	slope outl tool clampArea
Constraints:	IsOfType(SlopeTool, tool)
	IsOfType(Outline, outl)
	IsOfType(ProcArea, clampArea)
	IsOfType(Slope, slope)
	NotSame(outl, clampArea)
Effects:	+processed(slope)
	-unprocessed(slope)
Preconditions:	+processed(outl)
	+clampTurn(clampArea)
	+toolHeld(tool)
	+subarea(slope, outl)
	+available(tool)

MachineThread

Arguments:	thr outl tool clampArea
Constraints:	IsOfType(ThreadTool, tool)
	IsOfType(Outline, outl)
	IsOfType(Thread, thr)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
Effects:	+processed(thr)
	-unprocessed(thr)
Preconditions:	+processed(outl)
	+clampTurn(clampArea)
	+toolHeld(tool)
	+subarea(thr, outl)
	+unprocessed(thr)
	+available(tool)

MachineUndercutH1LeftToolOutline

Arguments:	ucut outl tool clampArea s1
Constraints:	IsOfType(LeftRTool, tool)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
	IsOfType(Outline, outl)
	IsOfType(Undercut, ucut)
	IsOfType(WpieceSide1, s1)
Effects:	+processedUcutHalf1(ucut)
	-unprocUcutHalf1(ucut)
Preconditions:	+processed(outl)
	+toolHeld(tool)
	+clampTurn(clampArea)
	+subarea(ucut, outl)
	+available(tool)
	+neighbour(s1, clampArea)

MachineUndercutH1LeftToolSide

Arguments:	ucut outl tool clampArea s1
Constraints:	IsOfType(LeftRTool, tool)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
	IsOfType(Outline, outl)
	IsOfType(Undercut, ucut)
	IsOfType(WpieceSide1, s1)
Effects:	+processedUcutHalf1(ucut)
	-unprocUcutHalf1(ucut)
Preconditions:	+processed(outl)
	+toolHeld(tool)
	+clampTurn(s1)
	+subarea(ucut, outl)
	+available(tool)
	+neighbour(s1, clampArea)

MachineUndercutH1RightToolOutline

Arguments:	ucut outl tool clampArea s2
Constraints:	IsOfType(RightRTool, tool)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
	IsOfType(Outline, outl)
	IsOfType(Undercut, ucut)
	IsOfType(WpieceSide2, s2)
Effects:	+processedUcutHalf1(ucut)
	-unprocUcutHalf1(ucut)
Preconditions:	+processed(outl)
	+toolHeld(tool)
	+clampTurn(clampArea)
	+subarea(ucut, outl)
	+available(tool)
	+neighbour(s2, clampArea)

MachineUndercutH1RightToolSide

Arguments:	ucut outl tool clampArea s2
Constraints:	IsOfType(RightRTool, tool)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
	IsOfType(Outline, outl)
	IsOfType(Undercut, ucut)
	IsOfType(WpieceSide2, s2)
Effects:	+processedUcutHalf1(ucut)
	-unprocUcutHalf1(ucut)
Preconditions:	+processed(outl)
	+toolHeld(tool)
	+clampTurn(s2)
	+subarea(ucut, outl)
	+available(tool)
	+neighbour(s2, clampArea)

MachineUndercutH2LeftToolOutline

Arguments:	ucut outl tool clampArea s2
Constraints:	IsOfType(LeftRTool, tool)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
	IsOfType(Outline, outl)
	IsOfType(Undercut, ucut)
	IsOfType(WpieceSide2, s2)
Effects:	+processedUcutHalf2(ucut)
	-unprocUcutHalf2(ucut)
Preconditions:	+processed(outl)
	+toolHeld(tool)
	+clampTurn(clampArea)
	+subarea(ucut, outl)
	+available(tool)
	+neighbour(s2, clampArea)

MachineUndercutH2LeftToolSide

Arguments:	ucut outl tool clampArea s2
Constraints:	IsOfType(LeftRTool, tool)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
	IsOfType(Outline, outl)
	IsOfType(Undercut, ucut)
	IsOfType(WpieceSide2, s2)
Effects:	+processedUcutHalf2(ucut)
	-unprocUcutHalf2(ucut)
Preconditions:	+processed(outl)
	+toolHeld(tool)
	+clampTurn(s2)
	+subarea(ucut, outl)
	+available(tool)
	+neighbour(s2, clampArea)

MachineUndercutH2RightToolOutline

Arguments:	ucut outl tool clampArea s1
Constraints:	IsOfType(RightRTool, tool)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
	IsOfType(Outline, outl)
	IsOfType(Undercut, ucut)
	IsOfType(WpieceSide1, s1)
Effects:	+processedUcutHalf2(ucut)
	-unprocUcutHalf2(ucut)
Preconditions:	+processed(outl)
	+toolHeld(tool)
	+clampTurn(clampArea)
	+subarea(ucut, outl)
	+available(tool)
	+neighbour(s1, clampArea)

MachineUndercutH2RightToolSide

Arguments:	ucut outl tool clampArea s1
Constraints:	IsOfType(RightRTool, tool)
	IsOfType(ProcArea, clampArea)
	NotSame(outl, clampArea)
	IsOfType(Outline, outl)
	IsOfType(Undercut, ucut)
	IsOfType(WpieceSide1, s1)
Effects:	+processedUcutHalf2(ucut)
	-unprocUcutHalf2(ucut)
Preconditions:	+processed(outl)
	+toolHeld(tool)
	+clampTurn(s1)
	+subarea(ucut, outl)
	+available(tool)
	+neighbour(s1, clampArea)

MakeToolHolderFree

Arguments:	tool
Constraints:	IsOfType(Tool, tool)
Effects:	+toolHolderFree()
	-toolHeld(tool)
Preconditions:	+toolHeld(tool)

TapHole

Arguments:	hole outl aTool clampArea
Constraints:	IsOfType(TappingTool, aTool)
	IsOfType(Outline, outl)
	IsOfType(ProcArea, clampArea)
	IsOfType(Hole, hole)
	NotSame(outl, clampArea)
Effects:	+processed(hole)
	-unprocessed(hole)
Preconditions:	+processed(outl)
	+clampNoTurn(clampArea)
	+toolHeld(aTool)
	+subarea(hole, outl)
	+available(aTool)

Appendix B

The Logistics Transportation Domain

Taken from (Veloso, 1994).

Object types: (8)

Airplane
Superclass: Carrier

Airport
Superclass: Location

Carrier
Superclass: None.

City
Superclass: None.

Location
Superclass: None.

Object
Superclass: None.

PostOffice
Superclass: Location

Truck
Superclass: Carrier

Predicates: (9)

atAirplane
Arguments: airplane loc

atObj
Arguments: obj loc

atTruck
Arguments: truck loc

diffCity
Arguments: loc1 loc2

insideAirplane
Arguments: obj plane

insideTruck
Arguments: obj airplane

locationAt
Arguments: loc city

sameCity
Arguments: loc1 loc2

truckInCity
Arguments: truck city

Operators: (6)

DriveTruck

Arguments:	truck locFrom locTo city
Constraints:	IsOfType(Truck, truck)
	IsOfType(Location, locFrom, locTo)
	IsOfType(City, city)
	NotSame(locFrom, locTo)
Purposes:	+atTruck(truck, locTo)
	-atTruck(truck, locFrom)
Preconditions:	+atTruck(truck, locFrom)
	+locationAt(locFrom, city)
	+sameCity(locFrom, locTo)
	+truckInCity(truck, city)
	+locationAt(locTo, city)

FlyAirplane

Arguments:	airplane locFrom locTo
Constraints:	IsOfType(Airplane, airplane)
	IsOfType(Airport, locFrom, locTo)
	NotSame(locFrom, locTo)
Purposes:	+atAirplane(airplane, locTo)
	-atAirplane(airplane, locFrom)
Preconditions:	+atAirplane(airplane, locFrom)
	+diffCity(locFrom, locTo)

LoadAirplane

Arguments:	obj airplane loc
Constraints:	IsOfType(Object, obj)
	IsOfType(Airplane, airplane)
	IsOfType(Airport, loc)
Purposes:	+insideAirplane(obj, airplane)
	-atObj(obj, loc)
Preconditions:	+atObj(obj, loc)
	+atAirplane(airplane, loc)

LoadTruck

Arguments:	obj truck loc
Constraints:	IsOfType(Object, obj)
	IsOfType(Truck, truck)
	IsOfType(Location, loc)
Purposes:	+insideTruck(obj, truck)
	-atObj(obj, loc)
Preconditions:	+atObj(obj, loc)
	+atTruck(truck, loc)

UnloadAirplane

Arguments:	obj airplane loc
Constraints:	IsOfType(Object, obj)
	IsOfType(Airplane, airplane)
	IsOfType(Airport, loc)
Purposes:	+atObj(obj, loc)
	-insideAirplane(obj, airplane)
Preconditions:	+insideAirplane(obj, airplane)
	+atAirplane(airplane, loc)

UnloadTruck

Arguments:	obj truck loc
Constraints:	IsOfType(Object, obj)
	IsOfType(Truck, truck)
	IsOfType(Location, loc)
Purposes:	+atObj(obj, loc)
	-insideTruck(obj, truck)
Preconditions:	+insideTruck(obj, truck)
	+atTruck(truck, loc)

Appendix C

The Artificial Domain ART-1D-RES

Object types: (0)

Predicates: (27)

G1
Arguments: None.
G10
Arguments: None.
G11
Arguments: None.
G12
Arguments: None.
G2
Arguments: None.
G3
Arguments: None.
G4
Arguments: None.
G5
Arguments: None.
G6
Arguments: None.
G7
Arguments: None.
G8
Arguments: None.
G9
Arguments: None.
I1
Arguments: None.
I10
Arguments: None.

I11
Arguments: None.
I12
Arguments: None.
I2
Arguments: None.
I3
Arguments: None.
I4
Arguments: None.
I5
Arguments: None.
I6
Arguments: None.
I7
Arguments: None.
I8
Arguments: None.
I9
Arguments: None.
occ-a
Arguments: None.
occ-b
Arguments: None.
RES-free
Arguments: None.

Operators: (16)

A1

Arguments:	None.
Constraints:	None.
Effects:	+G1()
Preconditions:	+I1()
	+occ-a()

A10

Arguments:	None.
Constraints:	None.
Effects:	+G10()
	-I9()
Preconditions:	+I10()
	+occ-b()

A11

Arguments:	None.
Constraints:	None.
Effects:	+G11()
	-I10()
Preconditions:	+I11()
	+occ-a()

A12

Arguments:	None.
Constraints:	None.
Effects:	+G12()
	-I11()
Preconditions:	+I12()
	+occ-b()

A2

Arguments:	None.
Constraints:	None.
Effects:	+G2()
	-I1()
Preconditions:	+I2()
	+occ-b()

A3

Arguments:	None.
Constraints:	None.
Effects:	+G3()
	-I2()
Preconditions:	+I3()
	+occ-a()

A4

Arguments:	None.
Constraints:	None.
Effects:	+G4()
	-I3()
Preconditions:	+I4()
	+occ-b()

A5

Arguments:	None.
Constraints:	None.
Effects:	+G5()
	-I4()
Preconditions:	+I5()
	+occ-a()

A6

Arguments:	None.
Constraints:	None.
Effects:	+G6()
	-I5()
Preconditions:	+I6()
	+occ-b()

A7

Arguments:	None.
Constraints:	None.
Effects:	+G7()
	-I6()
Preconditions:	+I7()
	+occ-a()

A8

Arguments:	None.
Constraints:	None.
Effects:	+G8()
	-I7()
Preconditions:	+I8()
	+occ-b()

A9

Arguments:	None.
Constraints:	None.
Effects:	+G9()
	-I8()
Preconditions:	+I9()
	+occ-a()

Alloc-a

Arguments:	None.
Constraints:	None.
Effects:	+occ-a()
	-RES-free()
Preconditions:	+RES-free()

Alloc-b

Arguments:	None.
Constraints:	None.
Effects:	+occ-b()
	-RES-free()
Preconditions:	+RES-free()

Free-a

Arguments:	None.
Constraints:	None.
Effects:	+RES-free()
	-occ-a()
Preconditions:	+occ-a()

Free-b

Arguments:	None.
Constraints:	None.
Effects:	+RES-free()
	-occ-b()
Preconditions:	+occ-b()

Bibliography

Aamodt, A. and Plaza, E. (1994). Case-based reasoning: Foundation issues, methodological variations and system approaches. *AI-Communications*, 7(1):pp 39–59.

Aha, D. W. and Goldstone, R. L. (1990). Learning attribute relevance in context in instance-based learning algorithms. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, pages pp 141–148, Cambridge, IN: Lawrence Erlbaum.

Althoff, K.-D., Auriol, E., Barletta, R., and Manago, M. (1995). *A Review of Industrial Case-Based Reasoning Tools.* AI Perspective Report, Oxford, UK: AI Intelligence.

Atkenson, C., Moore, A., and Schaal, S. (1995). Locally weighted learning. *Artificial Intelligence.*

Barrett, A. and Weld, D. (1994). Partial-order planning: Evaluating possible efficiency gains. 67(1):71–112.

Bergmann, R. and Wilke, W. (1995a). Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research*, 3:53–118.

Bergmann, R. and Wilke, W. (1995b). Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research*, 3:53–118.

Bhansali, S. and Harandi, M. (1994). When (not) to use derivational analogy: Lessons learned using apu. In Aha, D., editor, *Proceeding of AAAI-94 Workshop: Case-based Reasoning.*

Blumenthal, B. and Polster, B. (1994). Analysis and empirical studies of derivational analogy. *Artificial Intelligence.*

Blythe, J., Etzioni, O., Gil, Y., Joseph, R., Perez, A., Reilly, S., Veloso, M., and Wang, X. (1992). Prodigy4.0: The manual and tutorial. Technical report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University.

Cain, T., Pazzani, M. J., and Silverstein, G. (1991). Using domain knowledge to influence similarity judgement. In Kaufmann, M., editor, *Proceedings of the Case-Based Reasoning Workshop*, pages pp. 191–202, Washington, D.C.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, 12(3):231–272.

Etzioni, O. (1990). *A structural theory of explanation-based learning.* PhD thesis, Carnegie Mellon.

Etzioni, O. (1993a). Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62:255–301.

Etzioni, O. (1993b). A structural theory of explanation-based learning. *Artificial Intelligence*, 60:93–139.

Etzioni, O., Hanks, S., Karp, R. M., Madani, O., and Waarts, O. (1996). Efficient information gathering on the internet. In *Procceedings of FOCS-96.*

Fikes, R., Hart, P., and Nilsson, N. (1972). Learning and executing generalized robot plans. 3(4):251–288.

Fikes, R. and Nilsson, N. (1971). Strips: A new approach to the application of theorem proving in problem solving. 2:189–208.

Francis, A. G. J. and Ram, A. (1993). Computational models of the utility problem and their applications to a utility analysis of case-based reasoning. In *Proceedings of the Workshop on Knowledge Compilation and Speedup Learning (KCSL 93,ML-93).*

Francis, A. G. J. and Ram, A. (1995a). A comparative utility analysis of case-based reasoning and control-rule learning systems. In *Proceedings ECML-95*, number 912 in Lecture Notes in Artificial Intelligence.

Francis, A. G. J. and Ram, A. (1995b). A domain-independent algorithm for multi-plan adaptation and merging in least-commitment planning. In Aha, D. and Rahm, A., editors, *AAAI Fall Symposium: Adaptation of Knowledge Reuse*, Menlo Park, CA. AAAI Press.

Gil, Y. (1991). A specification of manufacturing processes for planning. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburg.

Golden, K. and Weld, D. (1996). Representing sensing actions: The middleground revisited.

Hammond, K. (1986). Chef: a model of case-based planning. In *Procceedings of American Asociation of Artificial Intelligence, AAAI-86.*

Hanks, S. and Weld, D. (1995). A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research*, 2.

Hayes, C. (1987). Using goal interactions to guide planning. pages 224–228.

Ihrig, L. and Kambhampati, S. (1994). Derivational replay for partial-order planning. In *Proceedings of AAAI-94*, pages 116–125.

Ihrig, L. and Kambhampati, S. (1996a). Design and implementation of a replay framework based on a partial order planner. In Weld, D., editor, *Proceedings of AAAI-96.* IOS Press.

Ihrig, L. and Kambhampati, S. (1996b). An explanation-based approach to improve retrieval in case-based planning. In Ghallab, M. and Milani, A., editors, *New Directions in AI Planning*. IOS Press.

Kambhampati, S. (1993). On the utility of systematicity: Understanding tradeoffs between redundancy and commitment in partial-order planning. pages 116–125.

Kambhampati, S. (1994). Expoiting causal structure to control retrieval and refitting during plan reuse. 10(2):213–244.

Kambhampati, S. (1996). Refinement planning: Status and prospectus. pages 1331–116. Invited Talk.

Kambhampati, S., Cutkosky, M., Tenenbaum, M., and Lee, S. (1991). Combining specialized reasoners and general purpose planners: A case study. pages 199–205.

Kambhampati, S., Ihrig, L., and Srivastava, B. (1996a). A candidate set based analysis of subgoal interactions in conjunctive goal planning. pages 125–133.

Kambhampati, S., Katukam, S., and Qu, Y. (1996b). Failure driven dynamic search control for partial order planners: An explanation-based approach. 88(1-2):253–315.

Karinthi, R., Nau, D., and Yang, Q. (1992). Handling feature interactions in process-planning. *Applied Artificial Intelligence*, 6:389–415.

Katukam, S. and Kambhampati, S. (1994). Learning explanation-based search control rules for partial order planning. pages 582–587.

Knoblock, C. (1996). Building a planner for information gathering: A report from the trenches. pages 134–141.

Koehler, J. (1994). Flexible plan reuse in a formal framework. In *Current Trends in AI Planning*, pages 171–184. IOS Press, Amsterdam, Washington, Tokio.

Kolodner, J. (1993). *Case-Based Reasoning*. Morgan Kaufmann Publishers.

Korf, R. (1987). Planning as search: A quantitative approach. 33:65–88.

Leake, D. B., editor (1996). *Case-Based Reasoning: Experiences, Lesons, and Future Directions*. AAAI Press/MIT Press.

McAllester, D. and Rosenblitt, D. (1991). Systematic nonlinear planning. pages 634–639.

Minton, S. (1988). *Learning Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, Boston.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–391.

Minton, S., Carbonell, J., Knoblock, C., Kuokka, D., Etzioni, O., and Y., G. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40:63–118.

Mitchell, T., Keller, R., and Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80.

Muñoz-Avila, H. and Hüllen, J. (1995). Retrieving relevant cases by using goal dependencies. In Veloso, M. and Aamodt, A., editors, *Case-Based Reasoning Research and Development, Proc. of the 1st Intern. Conference (ICCBR-95)*, number 1010 in Lecture Notes in Artificial Intelligence. Springer.

Muñoz-Avila, H. and Hüllen, J. (1996). Feature weighting by explaining case-based planning episodes. In *Third European Workshop (EWCBR-96)*, number 1168 in LNAI. Springer.

Muñoz-Avila, H., Paulokat, J., and Wess, S. (1994). Controlling non-linear hierarchical planning by case replay. In Keane, M., Halton, J., and Manago, M., editors, *Proc. of the 2nd European Workshop on Case-Based Reasoning (EWCBR-94).*

Muñoz-Avila, H., Paulokat, J., and Wess, S. (1995). Controlling non-linear hierarchical planning by case replay. In Keane, M., Halton, J., and Manago, M., editors, *Advances in Case-Based Reasoning. Selected Papers of the 2nd European Workshop (EWCBR-94)*, number 984 in Lecture Notes in Artificial Intelligence. Springer.

Muñoz-Avila, H. and Weberskirch, F. (1996a). Complete eager replay. In Sauer, J., Günter, A., and Hertzberg, J., editors, *Beiträge zum 10. Workshop 'Planen und Konfigurieren' (PuK-96).*

Muñoz-Avila, H. and Weberskirch, F. (1996b). Planning for manufacturing workpieces by storing, indexing and replaying planning decisions. In *Proc. of the 3nd International Conference on AI Planning Systems (AIPS-96).* AAAI-Press.

Muñoz-Avila, H. and Weberskirch, F. (1996c). A specification of the domain of process planning: Properties, problems and solutions. Technical Report LSA-96-10E, Centre for Learning Systems and Applications, University of Kaiserslautern, Germany.

Muñoz-Avila, H. and Weberskirch, F. (1997a). A case study on the mergeability of cases with a partial-order planner. In Steel, S., editor, *Proceedings of the Third European Conference on AI Planning (ECP-97).*

Muñoz-Avila, H. and Weberskirch, F. (1997b). Looking at features within a context from a planning perspective. In Aha, D. and Wettschreck, D., editors, *Proceedings of the ECML-97 MLNet Workshop Case-Based Learning: Beyond Classification of feature Vectors.*

Muñoz-Avila, H., Weberskirch, F., and Roth-Berghofer, T. (1997). On the relation between the context of a feature and the domain theory in case-based planning. In Leake, D. and Plaza, E., editors, *Proc. of the 2nd Intern. Conference on Case-based Reasoning (ICCBR-97)*, LNCS/LNAI. Springer.

Nau, D., Gupta, S., and Regli, W. (1995). AI planning versus manufacturing-operation planning: A case study.

Newell, A. and Simon, H. (1963). Gps: a program that simulates human thought. In Feigenbaum, E. and Feldman, J., editors, *Computers and Thought*. McGraw-Hill, New York.

Paulokat, J. and Wess, S. (1994). Planning for machining workpieces with a partial-order nonlinear planner. In Gil, Y. and Veloso, M., editors, *AAAI-Working Notes 'Planning and Learning: On To Real Applications'*, New Orleans.

Petrie, C. (1991a). Context maintenance. In *Proceedings of AAAI-91*, pages 288–295.

Petrie, C. (1991b). *Planning and Replanning with Reason Maintenance*. PhD thesis, University of Texas at Austin, Computer Science Dept.

Petrie, C. (1992). Constrained decision revision. In *Proceedings of AAAI-92*, pages 393–400.

Pryor, L. (1995). Decisions, decision: Knowledge goals in planning. In Hallam, J., editor, *Hybrid problems, hybrid solutions (Proc. of AISB-95)*, pages 181–192. IOS Press.

Ricci, F. and Avesani, P. (1995). Learning a local similarity metric for case-based reasoning. In *Case-Based Reasoning Research and Development, Proc. of the 1st International Conference (ICCBR-95)*, Sesimbra, Portugal. Springer Verlag.

Richter, M. (1995). The knowledge containers in similarity measures. Slides of Invited Talk at the First International Conference of Case-Based Reasoning (ICCBR-95). Available at http://wwwagr.informatik.uni-kl.de/ lsa/CBR/Richtericcbr95remarks.html.

Richter, M. M. (1992). *Prinzipien der Künstlichen Intelligenz (english: Principles of Artificial Intelligence)*. B. G. Teubner Sttutgart.

Roth-Berghofer, T. (1996). Explanation-based learning of control information from failures in planning. Masters thesis (in german), University of Kaiserslautern.

Russell, S. and Norvig, P. (1996). *Artificial Intelligence - A Modern Approach*. Prentice-Hall International.

Salzberg, S. L. (1991). A nearest hyperrectangle learning method. *Machine Learning*, 1.

Selberg, E. and O.Etzioni (1995). Multi-service search and comparison using the metacrawler. In *Proceedings of the 3nd International World-Wide Web Conference*.

Skalak, D. (1992). Representing cases as knowledge sources that apply local similarity metrics. In *Proceedings of the fourteenth Annual Conference of the Cognitive Science Society*, pages pp 325–330, Bloomington, IN: Lawrence Erlbaum.

Smyth, B. and Keane, M. (1994). Retrieving adaptable cases. Number 837 in LNAI. Springer.

Turney, P. D. (1996). The identification of context-sensitive features: A formal definition of context for concept learning. In *Proceedings of the ECML-96 Workshop on Learning in Context-Sensitive Domains*.

Veloso, M. (1994). *Planning and learning by analogical reasoning.* Number 886 in Lecture Notes in Artificial Intelligence. Springer Verlag.

Veloso, M. (1997). Nerge strategies for multiple case plan replay. In Leake, David B. Plaza, E., editor, *Proceedings of the Second Interantional Conference on Case-Based Reasoning, (ICCBR-97)*, volume LNCS/LNAI 1266. Springer.

Veloso, M. and Blythe, J. (1994). Linkability: Examining causal link commitments in partial-order planning. pages 13–19.

Veloso, M. and Carbonell, J. (1993). Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. *Machine Learning*, 10.

Weberskirch, F. (1995). Combining SNLP-like planning and dependency-maintenance. Technical Report LSA-95-10E, Centre for Learning Systems and Applications, University of Kaiserslautern, Germany.

Weberskirch, F. and Muñoz-Avila, H. (1997). Advantages of types in partial-order planning. In Müller, M., Schumann, O., and Schumann, S., editors, *Beiträge zum 11. Workshop 'Planen und Konfigurieren' (PuK-97)*, number FR-1997-001 in FORWISS-REPORT. FORWISS.

Weberskirch, F. and Paulokat, J. (1995). CAPlan - ein SNLP-basierter Planungsassistent. In Biundo, S. and Tank, W., editors, *Beiträge zum 9. Workshop 'Planen und Konfigurieren' (PuK-95)*, number DKFI-D-95-01 in DFKI-Dokument. DFKI.

Wettschereck, D. and Aha, D. W. (1995). Weighting features. In *Case-Based Reasoning Research and Development, Proc. of the 1st International Conference (ICCBR-95)*, number 1010 in Lecture Notes in Artificial Intelligence. Springer Verlag.

Wettschereck, D. and Diettrich, T, G. (1994). An experimental comparision of nearest neighbour and nearest hyperrectangle algorithms. *Machine Learning.*

Yang, H. and Lu, W. (1994). Case adaptation in a case-based process planning system. In Hammond, K., editor, *Proc. of the 2nd International Conference on AI Planning Systems (AIPS-94).* The AAAI Press.

Yang, Q., Nau, D., and Hendler, J. (1992). Merging separately generated plans with restricted interactions. 8(2):648–676.

Glossary

Same(<var>,<var'>), codesignation constraint, p. 22

NotSame(<var>,<var'>), non codesignation constraint, p. 22

IsOfType(<type>,<var>), type constraint, p. 22

IsNotOfType(<type>,<var>), type constraint, p. 22

+*predicateName(<var-1>, ..., <var-n>)*, precondition or effect in the add-list of operators, p. 22

−*predicateName(<var-1>, ..., <var-n>)*, effect in delete-list of operators, p. 22

(I, G), problem description, p. 23

$< S, \rightarrow, \rightarrow_{CL}, B >$, partial-order plan, p. 26

$<_{\rightarrow}$, order induced by $\rightarrow$, p. 26

$s_1 \rightarrow p@s_2$, causal link, p. 26

$s_3 \overset{+}{\longleftrightarrow} (s_1 \rightarrow p@s_2)$, positive threat, p. 27

$s_3 \overset{-}{\longleftrightarrow} (s_1 \rightarrow p@s_2)$, negative threat, p. 27

establisher(g,P), establisher of g, p. 58

$PLAN(I, G)$, set of plans solving (I, G), p. 58

$PLAN_{\prec}(I, G)$, set of plans solving (I, G) consistent with $\prec$, p. 58

$(I, G, \prec)$, extended problem description, p. 59

$\prec$, ordering constraints, p. 59

$[G_1, ..., G_m]$, sequence of dependency classes, p. 64

$\omega_{i,C}$, feature weight, p. 90

$sim^{wg}(C, P)$, weighted similarity metric, p. 90

$I \bigcap_{\theta} I_C$, predicates common to I and I_C modulo θ, p. 91

$SIM^{wg}(C, P)$, weighted similarity relation, p. 91

$\triangle_{k^C, f^C}$, incremental factor, p. 91

$G - G'$, set subtraction, p. 100

$\omega_{i,Coll}$, feature weight, p. 107

$(w_1, w_2]$, a left-opened interval, p. 108

$P_2 \setminus P_1$, plan difference, p. 123

$\triangle$, incremental factor, p. 157

Index

Lebenslauf

Angaben zur Person

Name:	Héctor Muñoz Avila
Geburtsdatum:	30.April 1967
Geburtsort:	Santafé de Bogotá, Kolumbien
Familienstand:	ledig

Schulbildung

1972-1976	Grundschule in Santafé de Bogotá
1977-1983	Gymnasium in Santafé de Bogotá
1983	Abschluß: Abitur

Hochschulstudium

1984-1990	Informatik und Mathematik Studium an der Universität "Universidad de los Andes"
1989	Abschluß: Informatik
1990	Abschluß: Mathematik
1990-1991	Magister Studium an der Universität "Universidad de los Andes"
1991	Abschluß: Magister Informatik
1994-1998	Promotion Studium an der Universität Kaiserslautern

Beruf

1991-1993	Wissenschaftlischer Mitarbeiter des Forschungszentrums CIJUS an der Universität "Universidad de los Andes"
1991-1993	Lehrtätigkeiten an der Universität "Universidad de los Andes"
1997-1998	Lehrtätigkeiten an der Universität Kaiserslautern