Plan Abstraction with Change of Representation Language

Ralph Bergmann
e-mail: bergmann@informatik.uni-kl.de
Centre for Learning Systems and Applications
University of Kaiserslautern
Dept. of Computer Science
P.O.-Box 3049, D-67653 Kaiserslautern, Germany

Abstract

Abstraction is one of the most promising approaches to improve the performance of problem solvers. Abstraction by *dropping sentences* of a domain description – as used in most hierarchical planners – is known to be very representation dependent. To overcome these drawbacks, we propose a more general view of abstraction involving the *change of representation language*. We have developed a new abstraction methodology and a related sound and complete learning algorithm that allows the complete *change of representation language* of *planning cases* from concrete to abstract.

1 Introduction

Abstraction is one of the most challenging and also promising approaches to improve complex problem solving and it is inspired by the way how humans seem to solve problems. Giunchiglia and Walsh [Giunchiglia and Walsh, 1992] have presented a comprehensive formal framework for abstraction and a comparison of the different abstraction approaches from theorem proving, planning, and model based diagnosis. For hierarchical planning, Korf's model of abstraction in problem solving [Korf, 1987] allows the analysis of reductions in search caused by single and multiple levels of abstraction. He has shown that in the optimal case, abstraction can reduce the expected search time from exponential to linear. Knoblock [Knoblock, 1994] as well as Bacchus and Yang [Bacchus and Yang, 1994] have developed approaches to construct a hierarchy of abstraction spaces automatically from a given concrete-level problem solving domain. All these abstraction methods, however, rely on abstraction by *dropping sentences* of the domain description which is a kind of *homomorphic abstraction* [Holte *et al.*, 1995]. It has been shown that these kinds of abstractions are highly representation dependent. For two classical planning domains, different "natural" representations have been analyzed and it turns out that there are several representations for which the classical abstraction techniques do not lead to significantly improved problem solvers [Knoblock, 1994; Holte *et al.*, 1994; Holte *et al.*, 1995].

From a knowledge-engineering perspecitve, many different aspects such as simplicity, understandability, and maintainability must be considered when developing a domain represenation. Therefore, we assume that representations of domains are given by knowledge engineers and rely on representations which we consider most "natural" for certain kinds of problems. We think it would require a large effort from a knowledge engineer to develop a representation and we believe that it is often impossible to develop a representation which is appropriate from a knowledge-engineering perspective and which also allows efficient hierarchical problem solving based on dropping sentences.

We take these observations as the motivation to develop a more general model of abstraction in problem solving. As already pointed out by Michalski [Michalski, 1994], abstraction, in general, can be seen as switching to a completely new representation language in which the level of detail is reduced. In problem solving, such a new abstract representation language must consist of completely new sentences and operators and not only of a subset of the sentences and operators of the concrete language. We want to propose a method of abstraction which allows the *complete change of representation language* of a problem and a solution from concrete to abstract and vice versa, if the concrete and the abstract language are given. The concrete language can be designed according to the requirements of the domain and the knowledge engineer and the abstract language can be designed such that useful abstractions can be achieved.

Additionally, our approach follows the idea to use *experience* from previously solved problems, usually available as a set of *cases*. By the use of experience, we want to come to better abstract solutions than a hierarchical problem solver which plans from scratch.

2 Case Abstraction and Refinement

We propose to use *experience* given in the form of *concrete planning cases* and to abstract this experience for its reuse in new situations. Therefore, we need a powerful abstraction methodology which allows the introduction of a completely new abstract terminology at the abstract level. This makes it possible that useful abstract solutions can be expressed for domains in which abstraction by dropping conditions is not sufficient. In particular, this methodology must not only serve as a means to analyze different abstraction approaches, but it must allow efficient algorithms for abstracting and refining problems and solutions.

As a prerequisite, this approach requires that the abstract language itself (state description and operators) is given by a domain expert in addition to the concrete level description. We also require that a set of admissible ways of abstracting states is implicitly predefined by a generic abstraction theory. This is of course an additional knowledge engineering requirement, but we feel that this is a price we have to pay to enhance the power of hierarchical problem solving.

In our approach to problem solving, we distinguish between a *learning phase* and a *problem solving phase*. During the learning phase, a set of abstract planning cases is generated from each available concrete case. An abstract planning case consists of an abstracted problem description together with an abstracted solution. The case abstraction procedure guarantees that the abstract solution contained in an abstract case can always be refined to become a solution of the concrete problem contained in the concrete case that became abstracted. Different abstract cases may be situated at different levels of abstraction or may be abstractions according to different abstraction aspects. Different abstract cases can be of different utility and can reduce the search space at the concrete level in different ways. It can also happen that several concrete cases share the same abstraction. The set of all abstract planning cases that are learned is organized in a case-base for efficient retrieval during problem solving.

During the problem solving phase, this case base is searched until an abstract case is found which can be applied to the current problem in hand. An abstract case is applicable to the current problem if the abstracted problem contained in the abstract planning case is an abstraction of the current problem. However, we cannot guarantee that an abstract solution contained in a selected abstract case can really be refined to become a solution to the current problem. It is at least known that each abstract solution from the case base was already useful for solving one or more previous problems, i.e. the problems contained in those concrete cases from which the abstract case was learned. Since the new problem is similar to these previous problems because both can be abstracted in the same way, there is at least a high chance that the abstract solution is also useful for solving the new problem. When the new problem is solved by refinement a new concrete case arises which can be used for further learning.

This approach is realized in PARIS (**P**lan **A**bstraction and **R**efinement in an **I**ntegrated **S**ystem), which is fully implemented and tested in the domain of process planning in mechanical engineering. Figure 1 shows an overview of the whole system and its components. Besides case abstraction and refinement, PARIS also includes an explanation-based approach for *generalizing cases* during learning and for *specializing* them during problem solving (see [Bergmann, 1992] for details). Furthermore, the system also includes additional mechanisms for evaluating different abstract and generalized cases according to computational effort required for *matching*, *specialization*, and *refinement* of cases. Based on this evaluation, several different indexing and retrieval mechanisms have been developed (see [Bergmann and Wilke, 1993] for details).

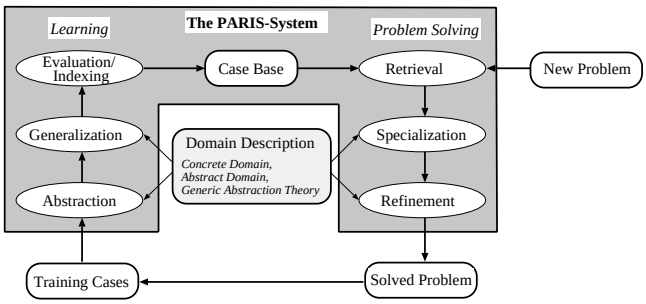

Figure 1: The Components of the PARIS-System

3 A Formal Model of Case Abstraction

This section presents a new formal model of case abstraction which allows to change the representation language of a planning case from concrete to abstract.

3.1 Basic Terminology

We use a STRIPS-like representation [Fikes and Nilsson, 1971] of operators and states to represent problem solving domains. A problem solving domain $\mathcal{D} = \langle \mathcal{L}, \mathcal{E}, \mathcal{O}, \mathcal{R} \rangle$ is described by a language $\mathcal{L}$, a set of essential atomic sentences [Lifschitz, 1987] $\mathcal{E}$ of $\mathcal{L}$, a set of operators $\mathcal{O}$ with related descriptions, and additionally, a set of Horn clauses $\mathcal{R}$ out of $\mathcal{L}$. A state $s \in \mathcal{S}$ describes the dynamic part of a situation in a domain and consists of a finite subset of ground instances of essential sentences of $\mathcal{E}$. With the symbol $\mathcal{S}$, we denote the set of all possible states descriptions in a domain, which is defined as $\mathcal{S} = 2^{\mathcal{E}^*}$, with $\mathcal{E}^* = \{e\sigma | e \in \mathcal{E}$ and σ is a substitution such that $e\sigma$ is ground$\}$. In addition, the Horn clauses $\mathcal{R}$ allow to represent static properties which are true in all situations. An operator $o(x_1, \ldots, x_n) \in \mathcal{O}$ is

described by a triple $\langle Pre_o, Add_o, Del_o \rangle$, where the precondition Pre_o is a conjunction of atoms of $\mathcal{L}$, and the add-list Add_o and the delete-list Del_o are finite sets of (possibly instantiated) essential sentences of $\mathcal{E}$. An instantiated operator o is *applicable* in a state s, if and only if $s \cup \mathcal{R} \vdash Pre_o$ holds. We write $s_1 \xrightarrow{o} s_2$ to denote that the operator o transforms a state s_1 into a state s_2. A *problem description* $p = \langle s_I, s_G \rangle$ consists of an initial state s_I together with a final state s_G. The problem solving task is to find a sequence of instantiated operators (a *plan*) $\bar{o} = (o_1, \ldots, o_l)$ which transforms the initial state into the final state $(s_I \xrightarrow{o_1} \cdots \xrightarrow{o_l} s_G)$. A *case* $C = \langle p, \bar{o} \rangle$ is a problem description p together with a plan $\bar{o}$ that solves p.

3.2 Case Abstraction

The problem of case abstraction can now be described as transforming a case which consists of a problem description and a correct solution (linear plan) from a concrete domain $\mathcal{D}_c$ into a case in an abstract domain $\mathcal{D}_a$ (see Figure 2). Therefore, we assume that in addition to the concrete problem solving domain $\mathcal{D}_c = \langle \mathcal{L}_c, \mathcal{E}_c, \mathcal{O}_c, \mathcal{R}_c \rangle$ an abstract problem solving domain $\mathcal{D}_a = \langle \mathcal{L}_a, \mathcal{E}_a, \mathcal{O}_a, \mathcal{R}_a \rangle$ is additionally supplied by a domain expert. In the remainder of this paper states and operators from the concrete domain are denoted by s^c and o^c respectively, while states and operators from the abstract domain are denoted by s^a and o^a respectively.

Because the abstract language can be explicitly stated by the user, this abstraction methodology allows to change the representation language of cases (problems and solutions) completely from concrete to abstract. Therefore, a user can define the level of abstraction and the required representation language on which reuse of solutions is most beneficial for a domain.

Case abstraction is a transformation that can reduce the level of detail of the description of the case with respect to two independent dimensions: first, the level of detail of the description of *individual states* can be reduced and second, the set of states that are kept in the abstract case can be reduced too. Formally, this transformation is decomposed into two independent mappings: a *state abstraction mapping* α, and a *sequence abstraction mapping* β [Bergmann and Wilke, 1995]. The state abstraction mapping transforms concrete state descriptions into abstract state descriptions, while the sequence abstraction mapping specifies which of the concrete states are mapped and which are skipped.

Definition 1 (State Abstraction Mapping) *A state abstraction mapping $\alpha : \mathcal{S}_c \rightarrow \mathcal{S}_a$ is a mapping from $\mathcal{S}_c$, the set of all states in the concrete domain, to $\mathcal{S}_a$, the set of all states in the abstract domain. In particular, α must be an effective total function.*

This general definition of a state abstraction mapping does not impose any restrictions on the kind of abstraction besides the fact that the mapping must be a total many-to-one function. However, to restrict the set of all possible state abstractions to a set of abstractions which a user considers useful, we assume that additional domain knowledge about how an abstract state relates to a concrete state can be provided. This knowledge must be expressed in terms of a domain specific *generic abstraction theory* $\mathcal{A}$.

Definition 2 (Generic Abstraction Theory) *A generic abstraction theory is a set of Horn clauses of the form $e_a \leftarrow a_1, \ldots, a_k$. In these rules e_a is an abstract essential sentence, i.e. $e_a = E_a\sigma$ for $E_a \in \mathcal{E}_a$ and a substitution σ. The body of a generic abstraction rule consists of a set of sentences from the concrete or abstract language, i.e. a_i are atoms out of $\mathcal{L}_c \cup \mathcal{L}_a$.*

Based on such a generic abstraction theory, we can restrict the set of all possible state abstraction mappings to those which are deductively justified by the generic abstraction theory.

Definition 3 (Justified State Abstraction) *A state abstraction mapping α is deductively justified by a generic abstraction theory $\mathcal{A}$, if the following conditions hold for all $s^c \in \mathcal{S}_c$:*

- *if $\phi \in \alpha(s^c)$ then $s^c \cup \mathcal{R}_c \cup \mathcal{A} \vdash \phi$ and*
- *if $\phi \in \alpha(s^c)$ then for all $\tilde{s}^c$ such that $\tilde{s}^c \cup \mathcal{R}_c \cup \mathcal{A} \vdash \phi$ holds, $\phi \in \alpha(\tilde{s}^c)$ is also fulfilled.*

In this definition the first conditions assures that every abstract sentence reached by the mapping is justified by the abstraction theory. Additionally, the second requirement guarantees that if an abstract sentence is used to describe an abstraction of one state, it must also be used to describe the abstraction of all other states, if the abstract sentence can be derived by the generic abstraction theory.

For the selection of those concrete states that have a corresponding abstraction, the sequence abstraction mapping is defined as follows:

Definition 4 (Sequence Abstraction Mapping) *A sequence abstraction mapping $\beta : \mathbb{N} \rightarrow \mathbb{N}$ relates an abstract state sequence $(s_0^a, \ldots, s_m^a)$ to a concrete state sequence $(s_0^c, \ldots, s_n^c)$ by mapping the indices $j \in \{1, \ldots, m\}$ of the abstract states s_j^a into the indices $i \in \{1, \ldots, n\}$ of the concrete states s_i^c, such that the following properties hold:*

- *$\beta(0) = 0$ and $\beta(m) = n$: The initial state and the goal state of the abstract sequence must correspond to the initial and goal state of the respective concrete state sequence.*
- *$\beta(u) < \beta(v)$ if and only if $u < v$: The order of the states defined through the concrete state sequence must be maintained for the abstract state sequence.*

Based on the two abstraction functions introduced, our intuition of case abstraction is captured in the following definition.

Definition 5 (Case Abstraction) *A case* $C_a = \langle\langle s_0^a, s_m^a\rangle, (o_1^a, \ldots, o_m^a)\rangle$ *is an abstraction of a case* $C_c = \langle\langle s_0^c, s_n^c\rangle, (o_1^c, \ldots, o_n^c)\rangle$ *with respect to the domain descriptions* $(\mathcal{D}_c, \mathcal{D}_a)$ *if* $s_{i-1}^c \xrightarrow{o_i^c} s_i^c$ *for all* $i \in \{1, \ldots, n\}$ *and* $s_{j-1}^a \xrightarrow{o_j^a} s_j^a$ *for all* $j \in \{1, \ldots, m\}$ *and if there exists a state abstraction mapping* α *and a sequence abstraction mapping* β*, such that:* $s_j^a = \alpha(s_{\beta(j)}^c)$ *holds for all* $j \in \{0, \ldots, m\}$.

This definition of case abstraction is demonstrated in Figure 2. The concrete space shows the sequence of n operations together with the resulting state sequence. Selected states are mapped by α into states of the abstract space. The mapping β maps the indices of the abstract states back to the corresponding concrete states.

In [Bergmann and Wilke, 1995] we showed that in this abstraction methodology hierarchies of abstraction spaces as well as different kinds abstraction can be handled simultaneously. Moreover, our abstraction methodology is more powerful than abstraction by dropping conditions as often realized in hierarchical planning [Sacerdoti, 1974; Knoblock, 1994].

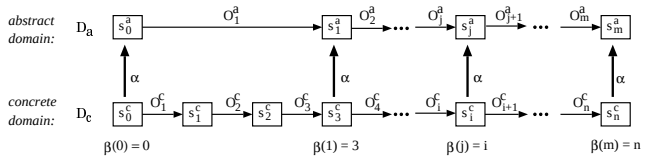

Figure 2: General idea of case abstraction

4 Computing Case Abstractions

We have developed an algorithm for automatically learning a set of abstract cases from a given concrete case. Thereby, we assume that the concrete domain $\mathcal{D}_c$ and the abstract domain $\mathcal{D}_a$ are given together with the generic abstraction theory. Please note that we do not require any kind operator abstraction rules. As input to the case abstraction algorithm, we assume a concrete case $C_c = \langle\langle s_0^c, s_n^c\rangle, (o_1^c, \ldots, o_n^c)\rangle$. The algorithm for case abstraction consists of four separate phases, which are sketched in the remained of this section. A detailed description of the algorihms for each phase can be found in [Bergmann and Wilke, 1995].

4.1 Phase-I: Computing the Concrete State Sequence

In the first phase, the sequence of concrete states which results from the sequential application of the operators in the concrete solution is computed. This leads to a sequence of states s_i^c, such that $s_{i-1}^c \xrightarrow{o_i^c} s_i^c$ holds for all $i \in \{1, \ldots, n\}$.

4.2 Phase-II: Deriving Abstract Essential Sentences

The second phase derives for each concrete state all possible abstract essential sentences that are justified by the generic abstraction theory. This derivation is performed by a SLD-refutation procedure which is sequentially applied to each concrete state. This leads to a sequence of abstract states $s_i^a = \{e \in \mathcal{E}_a | s_i^c \cup \mathcal{R}_c \cup \mathcal{A} \vdash e\}$ for all $i \in \{0, \ldots, n\}$. Now, each deductively justified state abstraction mapping α is restricted by $\alpha(s_i^c) \subseteq s_i^a$.

4.3 Phase-III: Computing Possible Abstract State Transitions

In the next phase we search for instantiated abstract operators which can transform an abstract state $\tilde{s}_i^a \subseteq s_i^a$ into a subsequent abstract state $\tilde{s}_j^a \subseteq s_j^a$ $(i < j)$. Therefore, the preconditions of the instantiated operator must at least be fulfilled in the state $\tilde{s}_i^a$ and consequently in also s_i^a. Furthermore, all added effects of the operator must be true in $\tilde{s}_j^a$ and consequently also in s_j^a.

The set of all possible operator transitions are collected as directed edges of a graph G which vertexes represent the abstract states (see Figure 3). For each pair of states (s_i^a, s_j^a) with $i < j$ it is checked whether there exists an operator o^a and a substitution σ of parameters of o^a such that $o^a(x_1, \ldots, x_u)\sigma$ is applicable in s_i^a, i.e. $s_i^a \cup \mathcal{R}_a \vdash Pre_{o^a}\sigma$ holds. Then the substitution σ is applied to the add-list of the operator leading to an instantiated add-list Add'_{o^a}. If $Add'_{o^a} \subseteq s_{aj}$ holds then instantiated operator derived thereby is included as a directed edge (from i to j) in the graph G.

By this algorithm it is guaranteed that each (instantiated) operator which leads from s_i^a to s_j^a is applicable in s_i^a and that all essential sentences added by this operator are contained in s_j^a. Furthermore, every instantiated operator which is applicable in s_i^a such that all essential sentences added by this operator are contained in s_j^a is also contained in the graph. From this follows immediately that if $\alpha(s_{\beta(i-1)}^c) \xrightarrow{o_i^a} \alpha(s_{\beta(i)}^c)$ holds for an arbitrary deductively justified state abstraction mapping α and a sequence abstraction mapping β, then the graph G contains a directed link from $\beta(i-1)$ to $\beta(i)$ which is marked with the operator o_i^a

4.4 Phase-IV: Determining Sound Paths

Based on the state abstractions s_i^a derived in phase-II and on the graph G computed in the previous phase, phase-IV selects a set of sound paths from the initial abstract state to the final abstract state. For this purpose, all complete paths from the initial to the final abstract state are determined. The abstract plan represented by each path is then checked whether it is a correct abstract plan in the abstract world, i.e. every operator is correctly

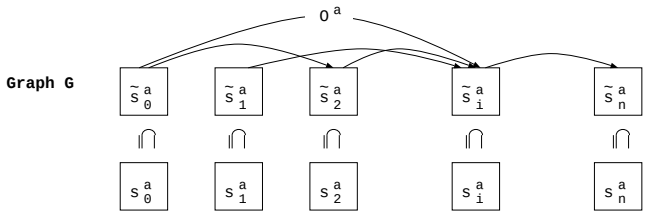

Figure 3: Graph of abstract operators

applicable in the state that results from the application of the previous abstract operator. Moreover, it must be checked that the state abstraction that results from the abstract plan is the same for all abstract states.

As a result, phase-IV returns all cases that are abstractions of the given concrete input case with respect to concrete and abstract domain definitions and the generic abstraction theory. Depending on the domain theory, more than a single abstract case can be learned from a single concrete case.

In [Bergmann and Wilke, 1995] we have described and analyzed this abstraction algorithm in detail and showed that it is sound and complete with respect to our abstraction methodology.

5 Retrieving and Refining Abstract Cases

Now we assume a case-base which contains a set of abstract cases that can be reused to solve a new problem. During problem solving, an abstract case must be selected from a case-base, and the abstract plan contained in this case must be refined to become a solution to the new problem.

To decide whether an abstract case can be *applied* to solve a concrete problem P, we have to determine a suitable state abstraction mapping. Because we assume deductively justified state abstraction mappings, the required state abstraction mapping α can always be induced by the set $\alpha^* = \bigcup_{i=0}^{m} s_i^a$. Consequently, C_a is applicable to the problem $p = \langle s_I^c, s_I^c \rangle$ if and only if $s_0^a = \{\Phi \in \alpha^* \mid s_I^c \cup \mathcal{R}_c \cup \mathcal{A} \vdash \Phi\}$ and $s_m^a = \{\Phi \in \alpha^* \mid s_G^c \cup \mathcal{R}_c \cup \mathcal{A} \vdash \Phi\}$.

The refinement of a selected abstract case starts with the concrete initial state from the problem statement. The search proceeds until a sequence of concrete operations is found which leads to a concrete state s^c, such that $s_1^a = \{\phi \in \alpha^* \mid s^c \cup \mathcal{R}_c \cup \mathcal{A} \vdash \phi\}$ holds. Since it is not guaranteed that the required concrete operator sequence exists, this search task may fail which causes the whole refinement process to fail also. If the first abstract operator can be refined successfully a new concrete state is found. This state can then be taken as a starting state to refine the next abstract operator in the same manner. If this refinement fails we can backtrack to the refinement of the previous operator and try to find an alternative refinement. If the whole refinement process reaches the final abstract operator it must directly search for an operator sequence which leads to the concrete goal state s_G^c. If this concrete goal state has been reached the concatenation of concrete partial solutions leads to a complete solution to original problem.

This refinement demands for a search procedure which allows an abstract goal specification. All kinds of forward-directed search such as depth-first iterative-deepening or best-first search procedures can be used for this purpose because states are explicitly constructed during search. These states can then be tested to see if they can be abstracted towards the desired goal. In PARIS we use depth-first iterative-deepening search. A detailed description of the refinement algorihm can be found in [Bergmann and Wilke, 1995].

This kind of refinement is different from the standard notion of refinement in hierarchical problem solving. This is because there is no strong correspondence between an abstract operator and a possible concrete operator. Moreover, the justification structure of a refined abstract plan is completely different from the justification structure of the abstract plan itself because of the completely independent definition of abstract and concrete operators. Even if this is a disadvantage compared to the usual refinement procedure used in hierarchical problem solving, the main computational advantage of abstraction caused by the decomposition of the original problem into smaller subproblems is maintained.

6 Experimental Evaluation in a Mechanical Engineering Domain

This section presents the example domain we have selected from the field of process planning in mechanical engineering.[1]

6.1 Process Planning for Rotary-Symmetric Workpieces

We have selected the goal of generating a process plan for the production of a rotary-symmetric workpiece on a lathe. The problem description contains the complete specification (especially the geometry) of the desired workpiece (goal state) together with a specification of the piece of raw material (called mold) it has to be produced from (initial state). A chucking fixture, together with the attached mold, is rotated with the longitudinal axis of the mold as rotation center. As the mold is rotated a cutting tool moves along some contour and thereby removes certain parts of the mold until the desired goal workpiece is produced. Within this process it is very hard to determine the sequence in which the specific parts of the workpiece have to be removed and the cutting tools to be used.

[1] This domain was adapted from the CaPlan-System [Paulokat and Wess, 1994], developed at the University of Kaiserslautern.

We have developed a representation of this domain which allows to handle a significant subset of relevant problems. In [Bergmann and Wilke, 1995] we present a detailed description of the domain, including all representation details. We also think that this representation is very "natural" for this domain.

6.2 An Example for Abstraction and Refinement

In Figure 4 we show how the PARIS system abstracts an example case and how it refines this abstract case to solve a different planning problem. The top of this figure shows a concrete planning case C_1. The representation of this case contains the exact geometrical specification of each element of the contour. Several areas of this contour are named by the indicated coordinates (e.g. #2, #2) for further reference in the cut-operations of the plan. The resulting abstract case is shown in the center of the figure. The abstract solution plan consists of a sequence of 6 abstract operators. The sequence of the operators in the plan is indicated by the Roman numerals. The particular abstraction is indicated between the concrete and the abstract case and denotes which sequence of concrete operators is turned into which abstract operator. The learned abstract case can now be

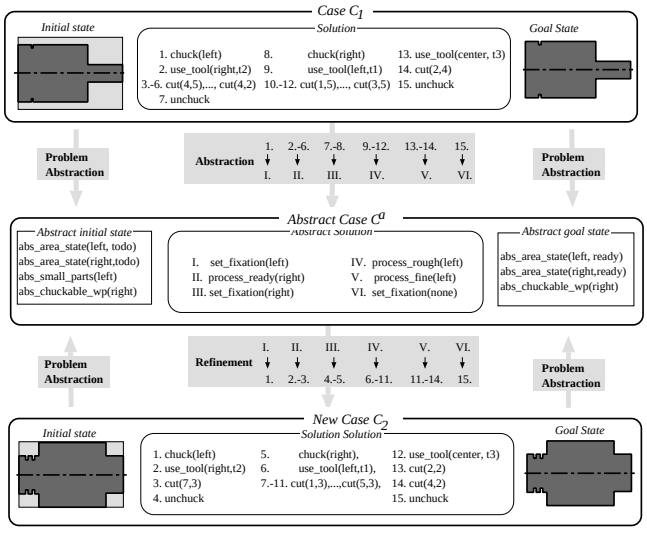

Figure 4: Abstracting and Refining an Example Case

used to solve the new problem C_2 whose initial and final concrete states are shown in the bottom of the figure. Even if the concrete workpiece looks quite different from the workpiece in case C_1 the abstract case can be used to solve the problem. We can see that most abstract operators (in particular the operators II, VI, and V) are refined to completely different sequences of concrete operators than those from which they were abstracted. This demonstrates the flexibility of this approach to plan reuse.

6.3 Experiments

We have performed a series of different experiments designed for evaluating PARIS. In a first experiment, we developed a domain description in the PRODIGY DESCRIPTION LANGUAGE which is equivalent to the conrete-level representation of the domain we used in PARIS. We applied the recent verion of ALPINE [Knoblock, 1994] to check whether abstraction by dropping sentences can improve problem solving in our domain. Unfortunately, for this representation, ALPINE was not able to generate an ordered monotonic abstraction hierarchy. The reason for this is that ALPINE can only distinguish a few different groups of literals because only a few different literal names (and argument types) could be used to represent the problem space. Therefore, which sentence to drop (or which criticalities to assign) cannot be decided statically by the name of the predicate or the type of the arguments.

The next experiment evaluates the PARIS approach. Table 1 summarizes some of the experimental results that were obtained by using PARIS to solve 100 different process planning problems. In this experiment, PARIS was trained with 5 different cases (10 cases in the second run) which were abstracted and stored for the subsequent problem solving. During problem solving, for each of the 100 problems, an abstract case is selected and refined. The table summarizes the percentage of problems that could be solved and the average computation time required to solve each problem. [2] The results are compared to those obtained in two separate experiments in which a) each problem is solved by pure search without abstraction and b) by hierarchical problem solving using our abstraction approach. Instead of retrieving the abstract solution from a case-base of abstract cases, each abstract solution is also constructed by search, similar to classical hierarchical problem solving.

Problem Solving Mode	Solved Problems	Solution Time
training with 5 cases	83 %	59 sec.
training with 10 cases	86 %	56 sec.
pure search	29 %	157 sec.
hierarchical planning	50 %	107 sec.

Table 1: Percentage of solved problems and average solution time.

We can see that abstraction by change of representation language can significantly improve the problem solving time as well as the number of problems that could be solved. We can also see that the reuse of cases leads to a more efficient problem solving than a hierarchical planner. This is an indication that the learned abstract

[2] We used a time limit of 200 CPU seconds on a SUN SPARC ELC computer for each problem. If this limit is exceded, the problem remains unsolved.

cases are more useful that abstract solutions created by search.

7 Requirements and Limitations

The most important prerequisite of this method is the availability of the required background knowledge, namely the concrete world description, the abstract world description, and the generic abstraction theory. For the construction of a planning system, the concrete world description must be acquired anyway, since it specifies the language of the problem description (essential sentences) and the problem solution (operators). The advantage of our approach is that the concrete world description can be designed according to the requirements of the domain and the knowledge engineer. It must not be designed in a way that a hierarchical planner can efficiently abstract the domain description automatically.

The abstract world and the generic abstraction theory must be acquired additionally. We feel that this is indeed the price we have to pay to use abstraction to make planning more tractable in certain practical situations. However, the abstract problem solving domain and the generic abstraction theory used have an important impact on the improvement in problem solving that can be achieved. Therefore, it is desirable to have a set of criteria which state how a "good" abstract domain definition should look. We now provide a set of four factors which determine the success of our approach.

7.1 Independent refinability of abstract operators

Following Korf's analysis of hierarchical problem solving [Korf, 1987], our plan refinement approach reduces the overall search space from b^n to $\sum_{i=1}^{m} b^{(\beta(i)-\beta(i-1))}$. Thereby, b is the average branching factor, n is the length of the concrete solution, and β is the sequence abstraction mapping used in the abstraction of the concrete case to the abstract case. As already mentioned, we cannot guarantee that an abstract plan which is applicable to a problem can really be refined. Furthermore, Korf's analysis assumes that no backtracking between the refinement of the individual abstract operators is required which cannot be guaranteed. Some of the computational advantage of abstraction is lost in either of these two cases.

However, if the abstract operators occuring in the abstract problem solving domain fulfill the strong requirement of *independent refinability*, then it is guaranteed that every applicable abstract case can be refined without any backtracking. The problem with this requirement is that it seems much to hard to develop an abstract problem solving domain in which all operators fulfill this requirement. Although we cannot expect that all operators in the abstract problem solving domain are independently refinable, a knowledge engineer developing an abstract domain should still try to define abstract operators which can be independently refined in *most* situations, i.e. for *most* s^c, $\tilde{s}^c \in \mathcal{S}_c$ and *most* state abstraction mapping α an applicable abstract operator can be refined to a concrete operator sequence. Although this notion of mostly independent refinability is not formal we feel that it is practically useful when developing an abstract domain definition.

7.2 Goal Distance of Abstract Operators

The goal distance (cf. subgoal distance [Korf, 1987]) is the maximum length of the sequence of concrete operators required to refine a particular abstract operator. The longer the goal distance the larger is the search space required to refine the abstract operator. In particular, the complexity of the search required to refine a complete abstract plan is determined by the largest goal distance of the abstract operators that occur in the abstract plan. Hence there is a good reason to keep the goal distance short. However, the goal distance negatively interacts with the next factor, namely the concrete scope of applicability of abstract operators.

7.3 Concrete Scope of Applicability of Abstract Operators

The concrete scope of applicability of an abstract operator specifies how many concrete states can be abstracted to an abstract state in which the abstract operator is applicable, and how many concrete states can be abstracted to an abstract state that can be reached by an abstract operator. This scope is determined by the definition of the abstract operator and by the generic abstraction theory which is responsible for specifying admissible state abstractions. The concrete scope of applicability of the abstract operators determines the applicability of the abstract plans that can be learned. An abstract plan which is only applicable to a few concrete problems is only of limited use in domains in which the problems to be solved vary very much. Hence, the concrete scope of applicability of abstract operators should be as large as possible. Unfortunately, according to our experience, abstract operators which have a large scope usually also have a larger goal distance and operators with a short goal distance don't have a large scope of applicability. Therefore, a compromise between these two contradicting issues must be found.

7.4 Complexity of the generic abstraction theory

The fourth factor which influences the problem solving time is the complexity of the generic abstraction theory. This theory must be applied each time a new concrete state is created during concrete level search. The more complex the generic abstraction theory, the more time is required to compute state abstractions. Hence, the generic abstraction theory should not require complicated inferences and should avoid backtracking within the SLD-refutation procedure.

8 Related Work

Most similar to the PARIS approach are the case-based planning systems Prodigy/Analogy [Veloso and Carbonell, 1993] and PRIAR [Kambhampati and Hendler, 1992]. However, these systems reuse planning cases directly and without doing any abstraction. Knoblock [Knoblock, 1994] presented an approach to automatically constructing abstraction hierarchies. This approach is limited to abstraction by dropping conditions and does not allow to change the representation language. It also does not allow to reuse cases. Our approach is also related to the idea of skeletal plans [Friedland and Iwasaki, 1985]. In the skeletal plan approach no model of the operators (neither concrete nor abstract) is used to describe the preconditions and effects of operators as is done in PARIS. There is no explicit notion of states and abstraction or refinement of states. Instead, the plan refinement is achieved by stepping down a hierarchy of operators, guided by heuristic rules for operator selection.

References

[Bacchus and Yang, 1994] F. Bacchus and Q. Yang. Downward refinement and efficiency of hierarchical problem solving. *Artificial Intelligence*, 71:43–100, 1994.

[Bergmann and Wilke, 1993] R. Bergmann and W. Wilke. Inkrementelles Lernen von Abstraktionshierarchien aus maschinell abstrahierten Plänen. In *Proceedings of the German GI Workshop on Machine Learning*. University of Karlsruhe, 1993.

[Bergmann and Wilke, 1995] R. Bergmann and W. Wilke. Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research* (to appear), 3:53–118, 1995. Also as Tech. Report LSA95-07E, Centre for Learning Systems and Applications, Univ. of Kaiserslautern.

[Bergmann, 1992] R. Bergmann. Knowledge acquisition by generating skeletal plans. In F. Schmalhofer, G. Strube, and Th. Wetter, editors, *Contemporary Knowledge Engineering and Cognition*, pages 125–133, Heidelberg, 1992. Springer.

[Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Friedland and Iwasaki, 1985] P. E. Friedland and Y. Iwasaki. The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, 1(2):161–208, 1985.

[Giunchiglia and Walsh, 1992] F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57:323–389, 1992.

[Holte *et al.*, 1994] R.C. Holte, C. Drummond, M.B. Perez, R.M. Zimmer, and A.J. MacDonald. Searching with abstractions: A unifying framework and new high-performance algorithm. In *Proceedings of the 10th Canadian Conference on Artificial Intelligence*, pages 263–270. Morgan Kaufmann Publishers, 1994.

[Holte *et al.*, 1995] R.C. Holte, T. Mkadmi, R.M. Zimmer, and A.J. MacDonald. Speeding up problem solving by abstraction: A graph-oriented approach. Technical report, University of Ottawa,, Ontario, Canada, 1995.

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

[Knoblock, 1994] C. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

[Korf, 1987] R. E. Korf. Planning as search: A quantitative approach. *Artifical Intelligence*, 33:65–88, 1987.

[Lifschitz, 1987] V. Lifschitz. On the semantics of strips. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, Timberline, Oregon, 1987.

[Michalski, 1994] R. Michalski. Inferential theory of learning as a conceptual basis for multistrategy learning. In R. Michalski and G. Tecuci, editors, *Machine Learning: A Multistrategy Approach*, number 11, chapter 1, pages 3–62. Morgan Kaufmann, 1994.

[Paulokat and Wess, 1994] J. Paulokat and S. Wess. Planning for machining workpieces with a partial-order, nonlinear planner. In *AAAI-Fall Symposium on Planning and Learning: On to Real Applications*, 1994.

[Sacerdoti, 1974] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Veloso and Carbonell, 1993] M. M. Veloso and J. G. Carbonell. Towards scaling up machine learning: A case study with derivational analogy in PRODIGY. In Steven Minton, editor, *Machine Learning Methods for Planning*, chapter 8, pages 233–272. Morgan Kaufmann, 1993.