# Learning Abstract Planning Cases

Ralph Bergmann and Wolfgang Wilke

University of Kaiserslautern,
Dept. of Computer Science,
P.O.-Box 3049, D-67653 Kaiserslautern, Germany
E-Mail: {bergmann,wilke}@informatik.uni-kl.de

**Abstract.** In this paper, we propose the PARIS approach for improving complex problem solving by learning from previous cases. In this approach, *abstract planning cases* are learned from given concrete cases. For this purpose, we have developed a new abstraction methodology that allows to completely *change the representation language* of a planning case, when the concrete and abstract languages are given by the user. Furthermore, we present a learning algorithm which is correct and complete with respect to the introduced model. An empirical study in the domain of process planning in mechanical engineering shows significant improvements in planning efficiency through learning abstract cases while an explanation-based learning method only causes a very slight improvement.

## 1 Introduction

Improving complex problem solving (i.e. planning, scheduling, design, or model-based diagnosis) by reusing past problem solving experience is one of the major topics addressed by machine learning research. Although a lot of methods for systematically solving complex problems are known from the literature on search, e.g. [20, 22] and planning [12, 38, 39, 45, 46, 26] most of them are intractable for solving problems in real-world applications due to basic search oriented nature of the algorithms. Learning from past experience promises to automatically acquire knowledge, useful to guide problem solvers so that they can improve their efficiency and competence. Most prominent are methods like explanation-based learning [33, 9, 37, 29, 31, 40, 10, 32, 23, 17] and analogical or case-based reasoning [7, 16, 42, 41]. While in explanation-based learning a control rule or a schema is *generalized* from an example problem solving trace, case-based approaches store detailed problem solving cases, index them appropriately and reuse and modify the cases according to the new problem to be solved.

In this paper we propose an alternative approach to improve complex problem solving. Instead of using learning methods which are based on *generalization*, we present a learning approach which computes *abstractions* of planning cases. As already pointed out by Michalski and Kodratoff [28, 27] abstraction has to be clearly distinguished from generalization. While generalization transforms a description along a set-superset dimension, abstraction transforms a description along a level-of-detail dimension. In general, abstraction requires *changing the complete representation language* while generalization usually maintains the representation language and introduces new variables for several objects to be generalized.

As the main contribution of this paper, we present an *abstraction methodology* and a related *learning algorithm* in which *abstract cases* are automatically derived from given concrete planning cases. Based on a given *concrete and abstract language* together with a *generic abstraction theory*, this learning approach allows to change the whole representation of a case from a concrete to abstract. Abstract cases learned from several concrete cases are then organized in a *case-base* for efficient retrieval during novel problem solving. This approach is realized in PARIS (**P**lan **A**bstraction and **R**efinement in an **I**ntegrated **S**ystem), which is fully implemented. PARIS is an integrated architecture for learning an problem solving [36] in which besides the abstraction mechanism described in this paper also an explanation-based approach is included. This allows to comprehensively investigate the different nature of abstraction and generalization as well as their integration.

The presentation of this approach is organized as follows. The next section describes the basic idea behind case abstraction and introduces the architecture of the PARIS system. The following three sections of the paper formalize the abstraction approach. After introducing the basic terminology, section 4 defines a new formal model of case abstraction. Section 5 gives a detailed description of a correct and complete learning algorithm for case abstraction. An experimental evaluation of the presented approach in a real-world domain is given in section 6. Finally, we discuss the presented approach in relation to similar work in the field.

## 2 Motivation: Case Abstraction and Refinement

This section wants to motivate the approach of learning abstract cases. The general approach is sketched and demonstrated by a real-world example. Furthermore, the PARIS architecture which realizes the presented approach is introduced.

### 2.1 Improving Problem Solving

Our main goal of learning is to improve the efficiency of a problem solver. We rely on the largely accepted view of problem solving which can be described as the task of transforming a given initial state into a given goal state by a sequence of available operators [34, 12, 8, 29, 41]. Thereby, initial state and goal state together constitute the problem description while the sequence of operators (plan) is the aspired solution. A definition of a problem solving domain usually consists of a description of the representation of the states which can occur during problem solving (usually a set of first order sentences) and a description of the available operators. An operator is usually described as a function which maps a certain starting state into a successor state, if certain conditions on the starting state hold. Efficiency is one of the major problems of this kind of problem solving because usually large search spaces need to be searched until a solution can be found.

## 2.2 The Basic Idea

Unlike well-known methods for improving problem solvers such as explanation-based learning [29, 31, 10, 32, 17] and analogical or case-based reasoning [7, 16, 42, 41] we propose an abstraction approach.

While the main goal of generalization is to extend the set of objects to which a certain piece of knowledge relates to, abstraction reduces the level of detail of a piece of knowledge. Unlike generalization, abstraction usually requires changing the representation language of an example or case [28, 15, 27] during learning. Several distinct concrete level objects need to be grouped into a smaller set of objects (out of the new representation language) at the abstract level. Abstraction must drop certain details of a description which are not useful for the kind of reasoning aspired. However, the most important relations between the concrete objects at the abstract level must be maintained. The advantage of abstraction is that it allows to simplify the representation and consequently speeds-up a reasoning process. Irrelevant details must not be considered anymore.

Our approach deals with the abstraction of planning cases. Thereby, a case consists of a problem description (initial state and goal state) and a related solution (operator sequence). The goal of abstraction is to reduce the level of detail of the problem description and solution in a consistent manner, i.e. the abstract solution must sill be a solution to the abstracted problem.

As a prerequisite, our approach requires that the abstract language and the concrete language are given by a domain expert. The abstract language itself is not constructed by the learning approach. This has the additional advantage that abstract cases are expressed in a language that the user is familiar with. Consequently, understandability and explainability, which are always important issues when applying a system, can be achieved much easier.

During a *learning phase*, a set of abstract cases is generated from each available concrete case. Different abstract cases may be situated at different levels of abstraction or may be abstractions according to different aspects. Usually, several concrete cases may share the same abstractions. The set of all abstract cases is the organized in a *case-base*.

When a new problem must be solved, the *problem solving phase* is entered. During this phase, the case base is searched until an abstract case is found which contains an abstract problem description which is an abstraction of the current concrete problem at hand. During further problem solving, the abstract solution (abstract plan) found in the retrieved abstract case must be refined (specialized) to become a solution to the current problem. During this process, this abstract solution serves as a *decomposition* of the original concrete problem into several smaller sub-problems, i.e. the sub-problems of refining the abstract steps. The sum of these sub-problems can usually be solved much more efficiently than the original problem as a whole [21, 19]. This effect leads to the desired overall improvement of problem solving.

### 2.3 A Real-World Example

To enhance the understanding of the following sections, we present an example now. This example is a simplification of the real-world domain of process planning in mechanical engineering.

**Domain Description.** As a real-world example domain we have selected a subtask from the field of process planning in mechanical engineering.[1] The goal is to generate 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). Rotary parts are manufactured by putting the mold into the fixture (chucking) of a lathe. The chucking fixture, together with the attached mold is then rotated with the longitudinal axis of the mold as rotation center. While 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 reached. Within this process, it is very hard to determine in which sequence the specific parts of the workpiece can be removed and which cutting tools must be used therefor. These decisions are very much influenced by the specific geometric shape of the workpiece.

**A Case.** In Figure 1, case $C_1$ shows an example of a rotary-symmetric workpiece which has to be manufactured out of a cylindrical mold.[2] The left side of the picture of case $C_1$ shows the drawing of the inital state (outer cylindrical form) together with the goal state (inner contour). The representation of this drawing 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. The right side of the case specifies the concrete plan which solves the problem. The solution plan consists of a sequence of 7 steps. In the first step, the workpiece (cylindrical mold) must be chucked at its left side. Then a cutting tool must be selected which can be used to cut the area specified by $(\#2, \#2)$ from the workpiece in the next step. In step 4, a different tool must be selected which allows to process the two small groves named $(\#1, \#2)$ and $(\#1, \#3)$. These groves are removed in step 5 and 6. Finally, the workpiece must be unchucked.

[1] This domain was adapted from the CaPlan-System [35], developed at the University of Kaiserslautern.

[2] Note, that this figure shows a 2-dimensional drawing of the 3-dimensional workpiece.

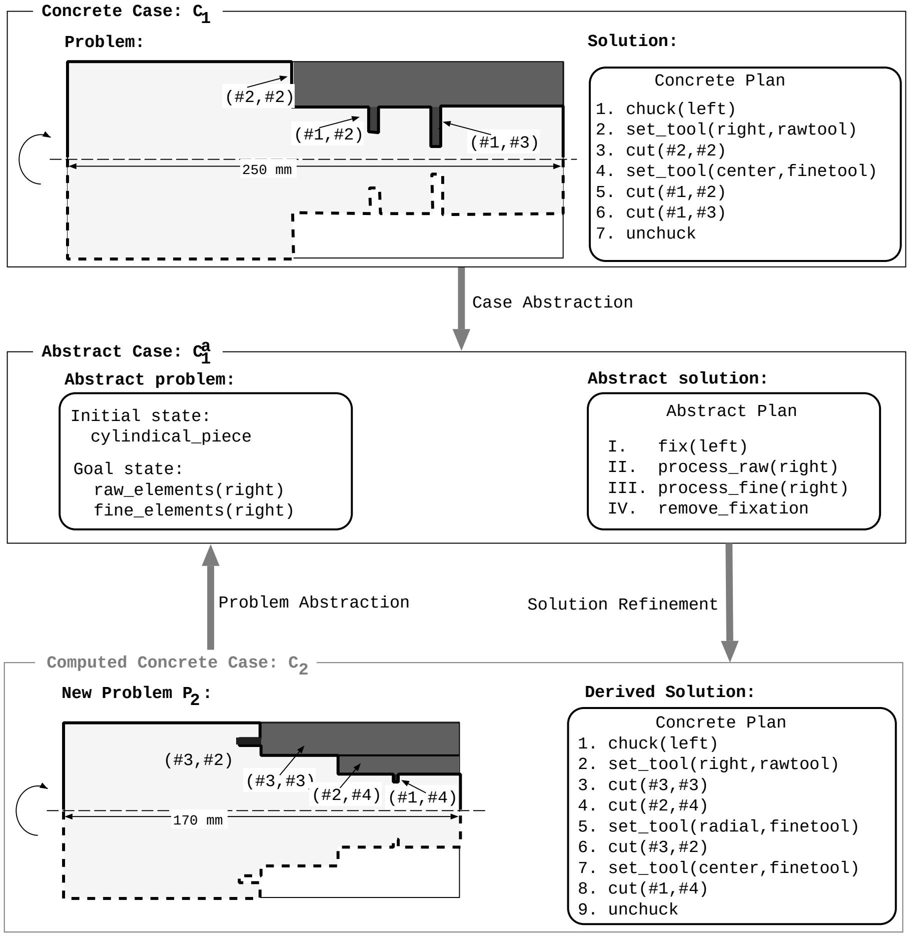


**Fig. 1.** Demonstration of the approach for a mechanical engineering example

**Case Abstraction.** The demonstrate the abstraction approach, we assume that case $C_1$ is available for learning. Case $C_1^a$ shows an abstraction which we may want to learn from $C_1$. The left side of this case shows a problem description with a reduced level of detail. The exact geometrical specification of the workpiece is omitted and replaced by qualitative description. The workpiece is divided into a left and a right side, and only raw and fine processing elements are distinguished. The right side of the description of case $C_1^a$ shows the abstracted plan which only consists of 4 abstract steps. The workpiece must be fixed at its left side, the raw parts on the right side must be processed, the fine parts on the right side must be processed, and finally the fixation must be removed. By this abstraction process, the concrete step 1 is turned into the abstract step I, the concrete steps 2 and 3 are turned into the abstract step II, the concrete steps 4,5, and 6 are abstracted towards step III, and step 7 is turned into the abstract step IV. Please note that in order to achieve the abstraction including a change of the representation language, the abstract language itself must be given in addition to the concrete language. In this example, the abstract language specifies how an abstract state can be described (e.g. my a term such as raw-elements(right)) and what abstract operators are available (e.g. process-raw) and how those abstract operators are specified.

**Problem Solving** The learned abstract case $C_1^a$ can be used to solve the new problem $P_2$ shown in the bottom of Figure 1. Although this problem is completely different at the concrete level from the problem in case $C_1$, it is identical at the abstract level. Both pieces have to be manufacture from a cylindrical mold even if the dimensions of the mold are quite different. Both goal pieces contain raw and fine elements on the right side of the workpiece. However, the detailed shape of those elements is completely different. Since the abstract problem as stated in the abstract cases $C_1^a$ matches the abstraction of the new problem $P_2$ completely, the abstract solution from $C_1^a$ can be used to solve the concrete problem. This abstract solution determines already the overall structure of the solution plan to be computed. Instead of solving the complete problem as a whole, the problem solver can now solve the four subproblems separately, i.e. determine a fixation, determine how to process the raw parts of the piece, determine how to process the fine parts of the piece, and finally determine how to remove the fixation. These four subproblems can be solved much more efficiently than the complete problem as a whole. For solving $P_2$, the abstract step II must be refined towards a sequence of the three concrete steps 2,3, and 4 as shown in the bottom of Figure 1. The abstract step III must be refined to a sequence of the four concrete steps 5,6,7, and 8.

Please note that except for the first two steps and the last step, the resulting concrete solution to problem $P_2$ is completely different from the solution contained in case $C_1$. However, an abstract case is still very helpful to find the new solution to the problem. Explanation-based or case-based approaches would not be able to learn knowledge from case $C_1$ to solve the problem $P_2$, because they cannot change the representation language appropriately. Neither a useful generalization can be derived nor can the case be reused directly.

### 2.4 The PARIS Architecture

PARIS (**P**lan **A**bstraction and **R**efinement in an **I**ntegrated **S**ystem) is a fully implemented system for learning and problem solving which realizes the sketched approach to case abstraction. Figure 2 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. Furthermore, the system includes several indexing and retrieval mechanisms for organizing and accessing the case-base of abstract cases, ranging from simple sequential search, via hierarchical clustering up to a sophisticated approach for balancing a hierarchy of abstract cases according to the statistic distribution of the cases within the problem space. This also includes methods for evaluating different abstract cases according to their ability to improve problem solving. More details on the generalization procedure can be found in [1], while the indexing and evaluation mechanisms are reported in [5, 44]. The whole multi-strategy system including the various interactions of the described components will be the topic of a forthcoming article, while first ideas can already be found [2, 4]. However, as the target of this paper we will concentrate on the core of PARIS, namely the abstraction approach. A detailed presentation of the related refinement approach can be found in [6].

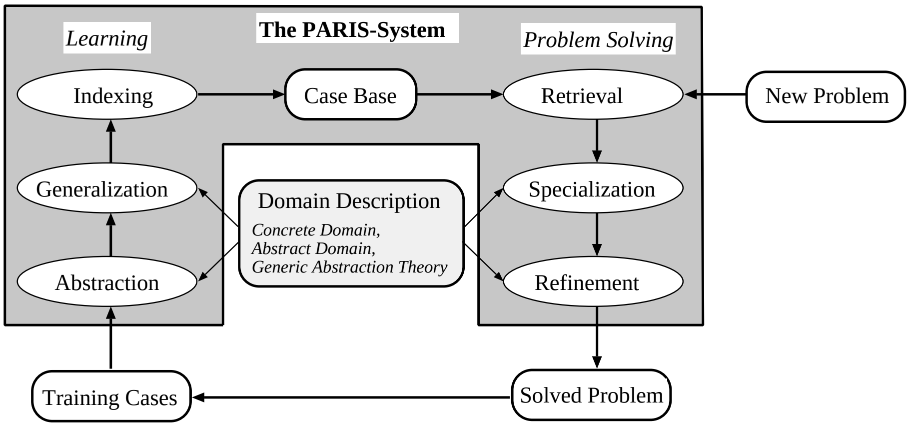


**Fig. 2.** The Components of the PARIS-System

## 3 Basic terminology

In this section we want to introduce the basic formal terminology followed throughout the rest of this paper. Following a STRIPS-oriented representation [12], the domain of problem solving $\mathcal{D} = \langle \mathcal{L}, \mathcal{E}, \mathcal{O}, \mathcal{R} \rangle$ is described by a language $\mathcal{L}$, a set of essential atomic sentences [24] $\mathcal{E}$ 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 $\sigma$ 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. These Horn clauses must not contain an essential sentence in the head of a clause.

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 $\mathcal{E}$. Furthermore, the variables occuring in the operator descriptions must follow the following restrictions: $\{x_1, \ldots, x_n\} \supseteq Var(Pre_o) \supseteq Var(Del_o)$ and $\{x_1, \ldots, x_n\} \supseteq Var(Add_o)$.[3]

An *instantiated* operator is an expression of the form $o(t_1, \ldots, t_n)$, with $t_i$ being ground terms of $\mathcal{L}$. For notational convenience, we define the *instantiated precondition* as well as the *instantiated add-list* and *delete-list* for an instantiated operator as follows: $Pre_{o(t_1,\ldots,t_n)} := Pre_o\sigma$, $Add_{o(t_1,\ldots,t_n)} := \{a\,\sigma | a \in Add_o\}$, $Del_{o(t_1,\ldots,t_n)} := \{d\,\sigma | d \in Del_o\}$, with $\langle Pre_o, Add_o, Del_o \rangle$ is the description of the (uninstantiated) operator $o(x_1, \ldots, x_n)$, and $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ the corresponding instantiation.

An instantiated operator $o$ is *applicable* in a state $s$ if and only if $s \cup \mathcal{R} \vdash Pre_o$ holds.[4] An instantiated operator $o$ transforms a state $s_1$ into a state $s_2$ (we write: $s_1 \xrightarrow{o} s_2$) if and only if $o$ is applicable in $s_1$ and $s_2 = (s_1 \setminus Del_o) \cup Add_o$. 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 $p$.

## 4 A Formal Model of Case Abstraction

In this section, we present a new formal model of case abstraction which allows to change the representation language of a case from concrete to abstract. As already stated, we assume, that in addition to the concrete language, the abstract language is supplied by a domain expert. Following the introduced formalism, we assume that the concrete level of problem solving is defined by a *concrete problem solving domain* $\mathcal{D}_c = \langle \mathcal{L}_c, \mathcal{E}_c, \mathcal{O}_c, \mathcal{R}_c \rangle$ and the abstract level of (case-based) problem solving is represented by an *abstract problem solving domain* $\mathcal{D}_a = \langle \mathcal{L}_a, \mathcal{E}_a, \mathcal{O}_a, \mathcal{R}_a \rangle$. 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. The problem of case abstraction can now be described as transforming a case from the concrete domain $\mathcal{D}_c$ into a case in the abstract domain $\mathcal{D}_a$ (see Figure 3). This transformation will now be formally decomposed into two independent mappings: a *state abstraction mapping* $\alpha$, and a *sequence abstraction mapping* $\beta$ [3].

---

[3] These restrictions can however be relaxed such that $\{x_1, \ldots, x_n\} \supseteq Var(Pre_o)$ is not required. But the introduced restriction simplifies the subsequent presentation.

[4] In the following, we will simply omit the parameters of operators and instantiated operators in case they are unambiguous or not relevant.

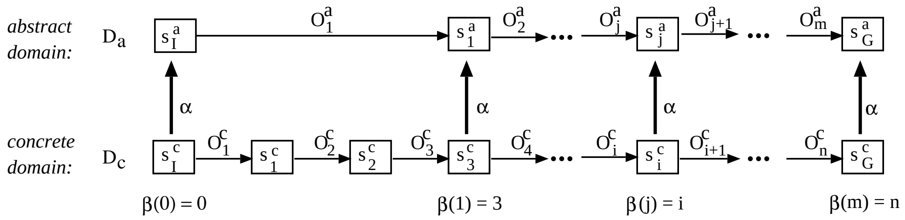


**Fig. 3.** General Idea of Abstraction

### 4.1 State Abstraction

A state abstraction mapping translates states of the concrete world into the abstract world. For this translation, we require additional domain knowledge about how an abstract state description relates to a concrete state description. We want to assume that this kind of knowledge can be provided in terms of a domain specific *generic abstraction theory* $\mathcal{A}$ [14]. In our model of case abstraction, such a generic abstraction theory defines each essential sentence $E_a \in \mathcal{E}_a$ in terms of the concrete domain by a set of horn-rules of the form $e_a \leftarrow a_1, \ldots, a_k$, where $e_a = E_a\sigma$ for a substitution $\sigma$ and $a_i$ are atoms out of $\mathcal{L}_c$.

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

**Definition 1 (Deductively Justified State Abstraction Mapping)**
*A deductively justified state abstraction mapping which is based on a generic abstraction theory $\mathcal{A}$, is a mapping $\alpha : \mathcal{S}_c \rightarrow \mathcal{S}_a$ for which the following conditions hold:*

- *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 considered in the abstraction of one state, it is also considered in the abstraction of all other states. Please note that a deductively justified state abstraction mapping can be completely induced, with respect to a generic abstraction theory, by a set $\alpha^* \subseteq \mathcal{E}_a^*$ as follows: $\alpha(s^c) := \{\phi \in \alpha^* | s^c \cup \mathcal{R}_c \cup \mathcal{A} \vdash \phi\}$.

To summarize, the state abstraction mapping transforms a concrete state description into an abstract state description and thereby changes the representation of a state from concrete to abstract.

### 4.2 Sequence Abstraction

The solution to a problem consists of a sequence of operators and a corresponding sequence of states. To relate an abstract solution to a concrete solution, the relationship between the abstract states (or operators) and the concrete states (or operators) must be captured. Thereby, each abstract state must have a corresponding concrete state, but not every concrete state must have an associated abstract state. To select those states of the concrete problem solution that have a related abstract state, the sequence abstraction mapping is defined as follows:

**Definition 2 (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.*

Note that the defined sequence abstraction mapping formally maps indices from the abstract domain into the concrete domain. However, an abstraction mapping should better map indices from the concrete domain to indices in the abstract domain such as the inverse mapping $\beta^{-1}$ does. But such a mapping is more inconvenient to handle formally, since the range of definition of $\beta^{-1}$ must always be considered. Therefore, we stick to presented definition.

### 4.3 Case Abstraction

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

**Definition 3 (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 $(\mathcal{D}_a, \mathcal{D}_c)$ 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 $\alpha$ and a sequence abstraction mapping $\beta$, such that: $s_j^a = \alpha(s_{\beta(j)}^c)$ for all $j \in \{0, \ldots, m\}$*

This definition of case abstraction is demonstrated in Figure 3. The concrete space shows the sequence of $n$ operations together with the resulting state sequence. Selected states are mapped by the $\alpha$ into states of the abstract space. The mapping $\beta$ maps the indices of the abstract states back to the corresponding concrete states.

In [6] we have discussed the generality of presented case abstraction methodology. We formally showed that hierarchies of abstraction spaces as well as abstractions with respect to different aspects can be represented using the presented methodology.

## 5 Computing Case Abstractions

Now we present the PABS-algorithm [3, 43] for automatically learning a set of abstract cases from a given concrete case. Thereby, we assume that a concrete domain $\mathcal{D}_c$ and an abstract domain $\mathcal{D}_a$ are given together with a generic abstraction theory $\mathcal{A}$.

Roughly speaking, the algorithm consists of four separate phases or sub-procedures. In the first sub-procedure, the sequence of concrete states which results from the execution of the concrete solution is computed. The second sub-procedure derives for each concrete state all possible abstract essential sentences justified by the generic abstraction theory. In the subsequent procedure, a graph of all applicable abstract operators is constructed, in which each edge leads from an abstract state to an abstract successor state. Finally, all consistent paths, starting at the abstract initial state and leading to the final abstract state are determined. Each of these paths represents a case which is an abstraction of the concrete case. In the following, we will present these phases in more detail. We presuppose a procedure for determining whether a conjunctive sentence in some language is a consequence of a set of clauses. More precisely, we assume a SLD-refutation procedure [25] which is given a set of clauses (a logic program) $C$ together with conjunctive sentence $G$ (a goal clause). The refutation procedure determines a set of answer substitutions $\Omega$ such that $C \vdash G\sigma$ for all $\sigma \in \Omega$. This set of answer substitutions is empty if $C \vdash G\sigma$ does not hold. We also require the derivation tree in addition to the answer substitutions. Then we write $\Pi = \mathrm{SLD}(C, G)$ and assume, that $\Pi$ is a set of pairs $(\sigma, \tau)$, where $\sigma$ is an answer substitution and $\tau$ is a derivation of $C \vdash G\sigma$.

### 5.1 Phase-I: Computing the Concrete State Sequence

As input to the case abstraction algorithm, we assume a concrete case $C_c = \langle\langle s_I^c, s_G^c\rangle, (o_1^c, \ldots, o_n^c)\rangle$. Note that $(o_1^c, \ldots, o_n^c)$ is a sequence of *instantiated* operators. In the first phase, the state sequence which results from the simulation of problem solution is computed as follows:

**Algorithm 1 (Phase-I: Computing the concrete state sequence)**
$s_0^c := s_I^c$
**for** $i := 1$ **to** $n$ **do**
  **if** $\mathrm{SLD}(s_{i-1}^c \cup \mathcal{R}_c,\ Pre_{o_i^c}) = \emptyset$ **then** *STOP "Failure: Operator not applicable"*
  $s_i^c := (s_{i-1}^c \setminus Del_{o_i^c}) \cup Add_{o_i^c}$
  **end**
**if** $s_G^c \not\subseteq s_n^c$ **then** *STOP "Failure: Goal state not reached"*

By this algorithm, the states $s_i^c$ are computed such that $s_{i-1}^c \xrightarrow{o_i^c} s_i^c$ holds for all $i \in \{1, \ldots, n\}$. If a failure occurs, the plan is not correct and the case is rejected.

## 5.2 Phase-II: Deriving Abstract Essential Sentences

Using the derived concrete state sequence as input, the following algorithm computes a sequence of abstract state descriptions ($s_i^a$) by applying the generic abstraction theory separately on each concrete state.

**Algorithm 2 (Phase-II: State abstraction)**

**for** $i := 0$ **to** $n$ **do**
  $s_i^a := \emptyset$
  **for each** $E \in \mathcal{E}_a$ **do**
    $\Omega := \mathrm{SLD}(s_i^c \cup \mathcal{R}_c \cup \mathcal{A}, E)$
    **for each** $\sigma \in \Omega$ **do**
      $s_i^a := s_i^a \cup \{E\,\sigma\}$
      **end**
    **end**
  **end**

Within the introduced model of case abstraction, we have now computed a superset for the outcome of possible state abstraction mappings. Each deductively justified state abstraction mapping $\alpha$ is restricted by $\alpha(s_i^c) \subseteq s_i^a = \{e \in \mathcal{S}_a | s_i^c \cup \mathcal{R}_c \cup \mathcal{A} \vdash e\}$ for all $i \in \{1, \ldots, n\}$. Consequently, we have determined all abstract sentences that an abstract case might require.

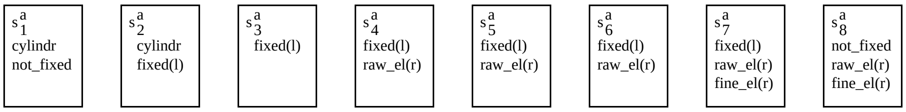


**Fig. 4.** An example of abstract states computed in phase-II. The abstract essential sentences are abbreviated as follows: *cylindr = cylindrical_piece, raw_el(r) = raw_elements(right), fine_el(r) = fine_elements(right), fixed(l) = fixed_piece(left).*

Figure 4 gives an example of the 8 abstract states $s_i^a$ computed during the abstraction of case $C^1$ shown in Figure 1. The abstract sentences used in these cases are a subset of the abstract sentences that occur in the abstract problem solving domain $D_a$ and which are defined in terms of the concrete sentences by the generic abstraction theory provided by the user.

## 5.3 Phase-III: Computing Possible Abstract State Transitions

In the next phase of the algorithm, 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 $s_j^a$.

**Algorithm 3 (Phase-III: Abstract state transitions)**

$G := \emptyset$<br>
**for** $i := 0$ **to** $n-1$ **do**<br>
  **for** $j := i+1$ **to** $n$ **do**<br>
    **for each** $o(x_1, \ldots, x_u) \in \mathcal{O}_a$ **do**<br>
      **let** $\langle Pre_o, Del_o, Add_o \rangle$ *be the description of* $o(x_1, \ldots, x_u)$<br>
      $\Pi := \mathrm{SLD}(s_i^a \cup \mathcal{R}_a, Pre_o)$<br>
      **for each** $\langle \sigma, \tau \rangle \in \Pi$ **do**<br>
        **let**$Add'_o = \{a\sigma | a \in Add_a\}$<br>
        (* Compute all possible instantiations *)<br>
        (* of added sentences which hold in $s_j^a$ *)<br>
        $M := \{\lambda\}$ *with* $\lambda = \emptyset$ *is the empty substitution.*<br>
        (* M is the set of possible substitutions *)<br>
        (* initially the empty substitution *)<br>
        **for each** $a \in Add'_o$ **do**<br>
          $M' := \emptyset$<br>
          **for each** $\theta \in M$ **do**<br>
            **for each** $e \in s_j^a$ **do**<br>
              **if** *there is a substitution* $\rho$ *such that* $a\theta\rho = e$ **then** $M' := M' \cup \{\theta\rho\}$<br>
              **end**<br>
            **end**<br>
          $M := M'$<br>
          **end**<br>
          (* Now, M contains the set of all possible substitutions *)<br>
          (* such that all added sentences are contained in $s_j^a$ *)<br>
          **for each** $\theta \in M$ **do**<br>
            $G := G \cup \{\langle i, j, o(x_1, \ldots, x_u)\sigma\theta, \tau \rangle\}$<br>
            **end**<br>
          **end**<br>
        **end**<br>
      **end**<br>
    **end**

The set of all possible operator transitions are collected as directed edges of a graph which vertexes represent the abstract states. In the algorithm, the set $G$ of edges of the acyclic directed graph is constructed. For each pair of states ($s_i^a$,$s_j^a$) with $i < j$ it is checked, whether there exists an operator $o(x_1, \ldots, x_u)\sigma$ which is applicable in $s_i^a$. For this purpose, the SLD-refutation procedure computes the set of all possible answer substitutions $\sigma$ such that the precondition of the operator is fulfilled in $s_i^a$. The derivation $\tau$ which belongs to each answer substitution is stored together with the operator in the graph since it is required for the next phase of case abstraction. This derivation is an "and-tree" where each inner-node reflects the resolution of a goal literal with the head of a clause and each leaf-node represents the resolution with a fact. Note that for proving the precondition of an abstract operator, the inner nodes of the tree always refer to clauses of the Horn rule set $\mathcal{R}_a$, while the leave-nodes represent facts stated in $\mathcal{R}_a$ or essential sentences contained in $s_i^a$. Then, each answer substitution $\sigma$ is applied to the add-list of the operator, leading to a partially instantiated add-list $Add'_o$. Note that there can still be variables in $Add'_o$ because the operator may contain variables which are not contained in its precondition but may occur in the add-list. Therefore, the set $M$ of all possible substitutions $\theta$ is incrementally constructed such that $a\theta \in s_{aj}$ holds for all $a \in Add'_o$. The completely instantiated operator derived thereby is finally included as a directed edge (from $i$ to $j$) in the graph $G$.

By the 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, if we claim that the applied SLD-refutation procedure is complete (it always finds all answer substitutions), then 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 $\alpha(s^c_{\beta(i-1)}) \xrightarrow{o_i^a} \alpha(s^c_{\beta(i)})$ holds for an arbitrary deductively justified state abstraction mapping $\alpha$ and a sequence abstraction mapping $\beta$, then $\langle \beta(i-1), \beta(i), o_i^a, \tau \rangle \in G$.

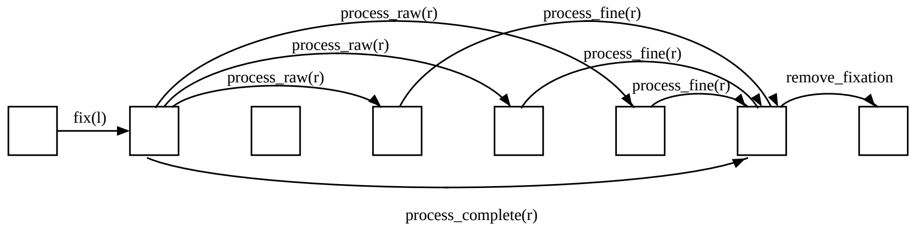


**Fig. 5.** An example of a transition graph computed in phase-III. The operator- parameters $l$ and $r$ abbreviate *left* and *right*, respectively.

Figure 5 gives an example of the graph $G$ computed in phase-III during the abstraction of case $C^1$ shown in Figure 1. The lables at the edges denote the respective instantiated operators that are determined by the algorithm.

### 5.4 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. During the construction of each path, a set of significant abstract sentences $\alpha^*$ and a sequence abstraction mapping $\beta$ are also determined. While the construction of the sequence abstraction mapping is obvious, the set $\alpha^*$ represents the image of a state abstraction mapping $\alpha$ and thereby determines the set of sentences that have to be reached in order to assure the applicability of the constructed operator sequence. Note that from $\alpha^*$ the state abstraction mapping $\alpha$ can be directly determined as follows: $\alpha(s_i^c) = \{e \in \alpha^* | s_i^c \cup \mathcal{R}_c \cup \mathcal{A} \vdash e\}$.

The idea of the algorithm is to start with an empty path. In each iteration of the algorithm, one path is extended by an operator from $G$. In this operator, new essential sentences $\alpha'$ may occur in the proof of the precondition or as added effects. The path constructed so far must still be consistent according to the extension of the state description and, in addition, the new operator must transform the sentences of $\alpha^*$ correctly.

**Algorithm 4 (Phase-IV: Searching sound paths)**
$Paths := \{\langle (), \emptyset, (\beta(0) = 0) \rangle\}$
**while** *it exists* $\langle (o_1^a, \ldots, o_k^a), \alpha^*, \beta \rangle \in Paths$ *with* $\beta(k) < n$ **do**
  $Paths := Paths \setminus \langle (o_1^a, \ldots, o_k^a), \alpha^*, \beta \rangle$
  **for each** $\langle i, j, o^a, \tau \rangle \in G$ *with* $i = \beta(k)$ **do**
    **let** $\tau_{\mathcal{E}}$ *be the set of essential sentences contained in the derivation* $\tau$
    **let** $\alpha' = \tau_{\mathcal{E}} \cup Add_{o^a} \cup \alpha^*$
    **if for all** $\nu \in \{1, \ldots, k\}$ *holds:*
      $(s^a_{\beta(\nu-1)} \cap \alpha') \xrightarrow{o^a_\nu} (s^a_{\beta(\nu)} \cap \alpha')$ **and**
      $(s^a_{\beta(k)} \cap \alpha') \xrightarrow{o^a} (s^a_j \cap \alpha')$ **then**
        $Paths := Paths \cup \{\langle (o_1^a, \ldots, o_k^a, o^a), \alpha', \beta \cup \{\beta(k+1) = j\} \rangle \}$
    **end**
  **end**
$Cases_{Abs} := \emptyset$
**for each** $\langle (o_1^a, \ldots, o_k^a), \alpha^*, \beta \rangle \in Paths$ *with* $\beta(k) = n$ **do**
  $Cases_{Abs} := Cases_{Abs} \cup \{\langle \langle s_0^a \cap \alpha^*, s_n^a \cap \alpha^* \rangle, (o_1^a, \ldots, o_k^a) \rangle\}$
  **end**
**return** $Cases_{Abs}$

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 will be learned from one concrete case.

For the abstraction of case $C^1$, the algorithm determines four consistent paths from the initial abstract state to the final abstract state. Three of these paths lead to one abstract case, namely the abstract case $C_1^a$ which had been already shown in Figure 1. The fourth path leads to a different abstract case which consists of the abstract operator sequence `fix(left), process_complete(right), remove_fixation`. This abstract cases differs from the other in that it is more abstract. The operator `process_complete` is an operator which completely processes one side of the workpiece, including raw and fine elements. This abstract cases is also valuable in situations in which no fine elements need to be processed for solving a problem.

### 5.5 Correctness, Completeness, and Complexity of the Algorithm

In [6] we have shown the strong connection between the formal model of case abstraction and the presented algorithm. We have proven that the algorithm is *correct*, that is every abstract case computed by the PABS algorithm is a case abstraction according to the introduced model. If the SLD-refutation procedure applied in PABS is *complete*, than every case which is an abstraction according to definition 3 is computed by PABS.

The complexity of the algorithm is mainly determined by the phases III and IV. The overall complexity of the complete PABS-algorithm is $O(n \cdot 2^{(n-1)})$ where $n$ is the length of the concrete level plan. The exponential factor comes from possibly exponential number of paths in a directed acyclic graph with $n$ nodes if every state is connected to every successor state. However, such a graph is really unrealistic in real applications. In particular, this worst-case complexity did not lead to a problem in our application domain.

## 6 Empirical Evaluation and Results

In this section, we want to present the results of an empirical study of the presented approach in the domain of mechanical engineering, introduced in section 2. This evaluation was performed with the fully implemented PARIS-system using the described abstraction component and the explanation-based plan generalization component (not described in this paper) separately for comparison. The problem solver is a depth-first iterative-deepening search procedure [20].

### 6.1 Experimental Setting

We have randomly generated a case base of 100 planning cases. From the 100 available cases, we have randomly chosen 10 training sets of 5 cases and 10 training sets of 10 cases. These training sets are selected independently from each other. For each training set, a related testing set is determined by choosing those of the 100 cases which are not used for training. By this procedure, training set and test set are completely independent. We trained PARIS with each of the training sets separately and measured the time for problem solving on the related testing sets. The time for *learning* a set of abstract cases from one concrete cases is between 30 and 180 seconds in our domain, depending on the length of the plan in the concrete cases. For *problem solving*, a time-bound of 200 CPU seconds was used for each problem. If the problem could not be solved within this time limit, the problem solver was aborted and the problem remained unsolved. The number of unsolved problems was also evaluated. For each training/testing set, PARIS was run in three different modes: a) using the abstraction approach, b) using the explanation-based generalization approach, and c) without any learning.

### 6.2 Results

Table 1 shows the average number of solved problems for the training sets of the two different sizes and the different modes of PARIS. These average numbers are computed from the 10 training and testing sets for each size. We can see that learning explanation-based generalizations slightly improves problem solving through a slightly increased number of solved problems that could be solved. Learning abstractions, however, leads to really significant improvements in the number of problems which could be solved and drastically outperforms the generalization approach.

**Table 1.** Percentage of solved problems after different training sets

| Size of training sets | Percentage of Solved Problems | | |
|---|---|---|---|
| (cases) | Abstraction | Generalization | No Learning |
| 5 | 83 | 37 | 29 |
| 10 | 86 | 37 | 29 |

A similar result can be found when examining the average time for solving a problem. Table 2 shows the average problem solving time for the different training sets in the three different modes of PARIS. Here, we can also see only a slight speed-up caused by learning generalizations but a much more significant speedup when the proposed abstraction approach is used.

**Table 2.** Problem solving time after different training sets

| Size of training sets | Average problem solving time (sec.) | | |
|---|---|---|---|
| (cases) | Abstraction | Generalization | No Learning |
| 5 | 59 | 142 | 156 |
| 10 | 56 | 141 | 156 |

Additionally, all of the above mentioned speedup results were analyzed with the maximally conservative sign test as proposed in [11]. When using abstraction, it turned out that 19 of the 20 training sets lead to highly significant speedups ($p < 0.0005$) of problem solving. Only one training set caused a significant speedup result for $p < 0.075$. Altogether, the reported experiment showed that even a small number of training cases (i.e. 5% and 10%) can already lead to strong improvements on problem solving in our domain.

Additionally, we made comparisons with ALPINE [18] which is part of PRODIGY [31] and automatically generates abstractions by dropping conditions. It turned out that for our domain, ALPINE was not able to improve problem solving at all because no useful abstractions could be built by dropping sentences as performed by ALPINE (see [6] for details).

# 7 Discussion

## 7.1 Conclusion

In this paper, we have presented a new approach to improving problem solving through reasoning from abstract cases. A methodology for abstracting planning cases and a sound and complete learning algorithm has been presented. An empirical evaluation in a real-world domain shows significant efficiency improvements of the problem solver while an explanation-based generalization approach leads to much slighter improvements.

Even if we have shown an advantage of abstraction over generalization in one specific application example, we clearly do not want to claim that abstraction is always better than generalization. In particular, explanation-based generalization has already proven useful in a couple of domains [29, 10, 40, 17]. However, the reason why it works so poorly in our domain is twofold. First of all, the representation of our domain requires a very detailed and complex representation of operators and states. Typically, a single concrete level state is described by over 200 ground facts. Moreover, the operator descriptions contain a large set of different preconditions. The well-known methods for explanation-based generalization of plans lead to rules or schemas which contain a very large set of conjunctively combined preconditions. This causes very high matching costs to determine the applicability of a rule or schema, because in the worst case the matching costs are exponential in the number (not on their size) of conditions. In our domain, these matching costs mostly exceed the savings caused by the application of the rules. Consequently, no overall speedup occurs. This problem is very much related to the utility-problem [30]. Our abstraction approach reduces the required level of detail in the descriptions that have to be matched to determine the applicability of an abstract case. Consequently, the matching process at the abstract level is much more efficient. However, the reused abstract plans must still be refined to come to a concrete solution, so that the potential speedup is obviously smaller than when reusing a concrete solution as in an explanation-based approach. But the key to success is that the matching costs at the abstract level are much smaller than the gain in efficiency.

The second reason for the advantage of abstraction over generalization in our domain is caused by the high flexibility of reusing abstract solutions. As already shown in section 2.3, an abstract case can be reused in a situation in which the concrete case or even a generalization of it is of no use. In particular if only a small number of cases is available for training, this flexibility is of high value.

Nevertheless, the availability of an adequate abstract domain theory is crucial to the success of the approach. This theory must allow to significantly simplify the representation of a case while the most important solution properties can be maintained. Otherwise, our abstraction approach would not work either. For the construction of a planning system, the concrete world descriptions must be acquired anyway, since they are the 'language' of the problem description (essential sentences) and the problem solution (operators). An appropriate abstract world and a generic abstraction theory must be acquired additionally. We feel that this is indeed the price we have to pay to make problem solving more tractable by learning in certain practical situations.

### 7.2 Related Work

**Theory of Abstraction.** Within Giunchiglia and Walsh's [15] theory of abstraction, the PARIS approach can be classified as follows: The formal system of the ground space $\Sigma_1$ is given by the concrete problem solving domain $\mathcal{D}_c$, and the abstract formal system $\Sigma_2$ is given by the language of the abstract problem solving domain $\mathcal{D}_a$. However, the operators of $\mathcal{D}_a$ are not turned into axioms of $\Sigma_2$. Instead, the abstract cases build the axioms of $\Sigma_2$. Moreover, the generic abstraction theory $\mathcal{A}$ defines the abstraction mapping $f : \Sigma_1 \Rightarrow \Sigma_2$. Within this framework, we can view PARIS as a system which learns useful axioms of the abstract system, by composing several smaller elementary axioms (the operators). However, to prove a formula (the existence of a solution) in the abstract system, exactly one axiom (case) is selected. So the deductive machinery of the abstract system is restricted with respect to the ground space. Depending on the learned abstract cases, the abstractions of PARIS are either theory decreasing (TD) or theory increasing (TI). If the case-base of abstract cases is completely empty, then no domain axiom is available and the resulting abstractions are consequently TD. But if the case-base contains the maximally abstract case $\langle\langle true, true\rangle(nop)\rangle$[5] then this case can be applied to every concrete problem and the resulting abstraction is consequently TI. Even if this maximally abstract case does not improve problem solving, it should be always included into the case-base to ensure the TI property, that is not losing completeness.

**Skeletal Plans.** The PARIS approach was inspired by the idea of skeletal plans [13]. A abstract cases can be seen as a skeletal plan, and our learning algorithm is a means to learn skeletal plans automatically out of concrete plans. Even if the idea of skeletal plans is intuitively very appealing, to our knowledge, this paper contains the first comprehensive experimental support of usefulness of planning with skeletal plans. Since we have shown that skeletal plans can be acquired automatically, this planning method can be applied more easily.

**ALPINE.** ALPINE [19, 18] automatically learns hierarchies of abstraction spaces from a given domain description or from a domain description together with a planning problem. ALPINE relies on abstraction by dropping sentences.

[5] *nop* is the 'no operation' operator which is always applicable and does not change the abstract state.

However, this enables ALPINE to generate abstraction hierarchies automatically. For the stronger abstraction framework we follow in PARIS, the automatic generation of abstraction hierarchies (or abstract domain descriptions) does not seem to be realistic due to the large (infinite) space of possible abstract spaces. To use our powerful abstraction methodology, we feel that we have to pay the price of losing the ability to automatically construct an abstraction hierarchy.

**Acknowledgements**
The authors want to thank Agnar Aamodt, Jaime Carbonell, Manuela Veloso, as well as Michael M. Richter and all members of our research group for helpful discussions and for remarks on earlier versions of this paper. We particularly appreciate the comments of the anonymous ECML reviewers. This research was partially funded by the Commission of the European Communities (ESPRIT contract P6322, the INRECA project). The partners of INRECA are AcknoSoft (prime contractor, France), tecInno (Germany), Irish Medical Systems (Ireland) and the University of Kaiserslautern (Germany).

## References

1. 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.
2. R. Bergmann. Learning abstract plans to speed up hierarchical planning. In P. Tadepalli, editor, *Proceedings of the ML92 Workshop on Knowledge Compilation and Speedup Learning*. University of Aberdeen, Scotland, 1992.
3. R. Bergmann. Learning plan abstractions. In H.J. Ohlbach, editor, *GWAI-92 16th German Workshop on Artificial Intelligence*, volume 671 of *Springer Lecture Notes on AI*, pages 187–198, 1992.
4. R. Bergmann. Integrating abstraction, explanation-based learning from multiple examples and hierarchical clustering with a performance component for planning. In Enric Plaza, editor, *Proceedings of the ECML-93 Workshop on Integrated Learning Architectures (ILA-93)*, Vienna, Austria, 1993.
5. 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.
6. R. Bergmann and W. Wilke. Building and refining abstract planning cases by change of representation language. Technical report, University of Kaiserslautern,, 1994.
7. J. G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise aquisition. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine learning: An artificial intelligence approach*, volume 2, chapter 14, pages 371–392. Morgan Kaufmann, Los Altos, CA, 1986.
8. D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.
9. G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.
10. O. Etzioni. A structural theory of explanation-based learning. *Artificial Intelligence*, 60:93–139, 1993.

11. O. Etzioni and R. Etzioni. Statistical methods for analyzing speedup learning. *Machine Learning*, 14:333–347, 1994.
12. 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.
13. P. E. Friedland and Y. Iwasaki. The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, 1(2):161–208, 1985.
14. A. Giordana, D. Roverso, and L. Saitta. Abstracting background knowledge for concept learning. In Y. Kodratoff, editor, *Lecture Notes in Artificial Intelligence: Machine Learning-EWSL-91*, pages 1–13, Berlin, 1991. Springer.
15. F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57:323–389, 1992.
16. S. Kambhampati and J. A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55:193–258, 1992.
17. S. Kambhampati and S. Kedar. A unified framework for explanation-based generalization of partially ordered partially instantiated plans. *Artificial Intelligence*, 67:29–70, 1994.
18. C. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.
19. C. A. Knoblock. A theory of abstraction for hierachical planning. In *Proceedings of the Workshop on Change of Representation and Inductive Bias*, pages 81–104, Boston, MA, 1989. Kluwer.
20. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artifical Intelligence*, 27:97–109, 1985.
21. R. E. Korf. Planning as search: A quantitative approach. *Artifical Intelligence*, 33:65–88, 1987.
22. R. E. Korf. Linear-space best-first search. *Artifical Intelligence*, 62:41–78, 1993.
23. P. Langley and J.A. Allen. A unified framework for planning and learning. In S. Minton, editor, *Machine Learning Methods for Planning*, chapter 10, pages 317–350. Morgan Kaufmann, 1993.
24. V. Lifschitz. On the semantics of strips. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, Timberline, Oregon, 1987.
25. J.W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.
26. D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 634–639, 1991.
27. 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.
28. R. S. Michalski and Y. Kodratoff. Research in machine learning: Recent progress, classification of methods, and future directions. In Y. Kodratoff and R. S. Michalski, editors, *Machine learning: An artificial intelligence approach*, volume 3, chapter 1, pages 3–30. Morgan Kaufmann, San Mateo, CA, 1990.
29. S. Minton. *Learning Search Control Knowledge: An Explanation-Based Approach*. Kluwer, Boston, MA, 1988.
30. S. Minton. Quantitativ results concerning the utility of explanation-based learning. *Artifical Intelligence*, 42:363–391, 1990.
31. S. Minton, J. G. Carbonell, C.A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

32. S. Minton and M. Zweben. Learning, planning and scheduling: An overview. In S. Minton, editor, *Machine Learning Methods for Planning*, chapter 1, pages 1–30. Morgan Kaufmann, 1993.
33. T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.
34. A. Newell and H. Simon. *Human Problem Solving*. Prentice-Hall Englewood Cliffs, NJ, 1972.
35. 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.
36. E. Plaza, A. Aamodt, A. Ram, W. VandeVelde, and M. vanSommeren. Integrated learning architectures. In B. Brazdil, editor, *European Conference on Machine Learning: ECML-93*, volume 667 of *Lecture Notes in Artificial Intelligence*, pages 429–441, Berlin, 1993. Springer.
37. P.S. Rosenbloom and J.E. Laird. Mapping explanation-based learning onto SOAR. In *Proceedings National Conference on Artificial Intelligence*, volume 2, Philadelphia, PA, August 1986.
38. E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.
39. E.D. Sacerdoti. *A Structure for Plans and Behavior*, volume 5. American-Elsevier, New York,, 1977.
40. J.W. Shavlik and P. O'Rorke. Empirically evluation ebl. In *Investigating Explanation-Based Learning*, volume 5, chapter 7, pages 222–294. Kluwer Academic Publishers, 1993.
41. M. M. Veloso. PRODIGY/ANALOGY: Analogical reasoning in general problem solving. In M.M. Richer, S. Wess, K.D. Althoff, and F. Maurer, editors, *Topics in Case-Based Reasoning*, pages 33–52. Lecture Notes in AI, Vol. 837, Springer, 1994.
42. 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.
43. W. Wilke. Entwurf und Implementierung eines Algorithmus zum wissensintensiven Lernen von Planabstraktionen nach der PABS-Methode. Projektarbeit, Universität Kaiserslautern, 1993.
44. W. Wilke. Entwurf, Implementierung und experimentelle Bewertung von Auswahlverfahren für abstrakte Pläne im fallbasierten Planungssystem PARIS. Master's thesis, University of Kaiserslautern, Germany, 1994.
45. D. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.
46. Q. Yang and J.D. Tenenberg. Abtweak: Abstracting a nonlinear, least commitment planner. In *Proceedings of the 8th National Conference on Aritificial Intelligence*, pages 204–209, Boston, MA, 1990.

This article was processed using the LaTeX macro package with LLNCS style