# Systematic Approach to the Design of Representation-Changing Algorithms

**Eugene Fink** *
Computer Science, Carnegie Mellon University
Pittsburgh, Pennsylvania 15213, USA
eugene@cs.cmu.edu

## Abstract

The performance of all problem-solving systems depends crucially on problem representation. The same problem may be easy or difficult to solve depending on the way we describe it. Researchers have designed a variety of learning algorithms that deduce important information from the description of the problem domain and use the deduced information to improve the representation. Examples of these representation improvements include generating abstraction hierarchies, replacing operators with macros, decomposing a problem into subproblems, and selecting primary effects of operators. There has, however, been little research on the common principles underlying the representation-improving algorithms and the notion of useful representation changes has remained at an informal level.

We present preliminary results on a systematic approach to the design of algorithms for automatically improving representations. We identify the main desirable properties of such algorithms, present a framework for formally specifying these properties, and show how to implement a representation-improving algorithm based on the specification of its properties. We illustrate the use of this approach by developing two novel algorithms that improve problem representations.

*This research is supported by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government.

## 1 Introduction

The *problem representation* in an AI problem-solving system is the input to the system. In most problem-solving systems, it includes the description of the operators in a problem domain, the initial and goal states of a problem, and possibly some other information, such as control rules and an abstraction hierarchy. The information given directly as an input is called *explicit*, and the information that can be deduced from the input is called *implicit*. Every problem representation leaves some information implicit.

The performance of all problem-solving systems depends crucially on problem representation. The same problem may be easy or difficult to solve depending on the way we describe it. Psychologists and AI researchers have accumulated much evidence of the importance of good representations for human problem solvers [Newell and Simon, 1972; Simon *et al.*, 1985; Kaplan and Simon, 1990] and AI problem-solving systems [Newell, 1966; Amarel, 1968; Korf, 1980].

Explicit representation of important information improves the performance of a problem-solver. For example, we may improve the efficiency of a problem-solving system by encoding useful information about the domain in control rules [Minton, 1988] and an abstraction hierarchy [Knoblock, 1994]. On the other hand, explicit representation of irrelevant information decreases efficiency: if we do not mark such information as unimportant for the problem, the system attempts to use it, which takes extra computation and may lead the system to explore useless branches of the search tree. For example, if we add unnecessary extra operators to the domain description, and if these operators may (but need not) be used in solving a problem, the branching factor of search increases and the efficiency decreases.

Different problem-solving algorithms use different information about the domain and, therefore, perform efficiently with different representations [Stone *et al.*, 1994]. There is no "universal" representation that works well with all algorithms. The task of finding a good representation is usually left to the human user.

Newell was first to discuss the role of representation in AI problem solving: he showed that the complexity of reasoning in some games and puzzles strongly depends on the representation [Newell, 1965; 1966]. Later, Newell with several other researchers implemented the Soar system [Laird *et al.*, 1987; Newell, 1992], capable of using different descriptions of a problem domain to facilitate problem solving and learning. Soar, however, does not generate new representations; the human user must provide all domain descriptions. A similar approach was used in the FERMI expert system [Larkin *et al.*, 1988], which automatically selects a representation for a given problem among several hand-coded representations.

Many researchers have addressed the representation problem by designing learning algorithms that deduce important information from the domain description and use the deduced information to improve the representation [Allen *et al.*, 1992; Carbonell, 1990]. Examples of these representation improvements include decomposing a problem into subproblems [Newell *et al.*, 1960], generating abstraction hierarchies [Knoblock, 1994; Bacchus and Yang, 1994], replacing operators with macros [Korf, 1985; Mooney, 1988; Shell and Carbonell, 1989], replacing problems with similar simpler problems [Hibler, 1994], and selecting primary effects of operators [Fink and Yang, 1992; 1993].

A generalized model of improving representations in problem solving was suggested by Korf, who formalized the concept of representation changes based on the notions of isomorphism and homomorphism of search spaces [Korf, 1980]. Korf's model, however, does not address "a method for evaluating the efficiency of a representation relative to a particular problem solver and heuristics to guide the search for an efficient representation for a problem" ([Korf, 1980], page 75), whereas the use of such heuristics is essential for developing an efficient representation-changing system.

Even though AI researchers implemented many systems for automatically improving problem representation, there has been little research on the common principles used in the design of representation-changing algorithms and methods for developing new algorithms. Because of the lack of techniques and guidelines for the design of representation-changing algorithms, implementing a new representation-changer is usually a complex research problem.

In this paper, we present preliminary results on a systematic approach to the design of representation-changing algorithms. We concentrate on efficiency-improving representation changes in general-purpose problem-solving systems, such as PRODIGY [Veloso *et al.*, 1995] and UCPOP [Penberthy and Weld, 1992], which represent solutions as sequences of operators.

We identify the main desirable properties of representation changing algorithms and describe a method for formally specifying these properties, which enables us to abstract the major decisions in the development of a representation-changer from implementational details. We show how to write a formal specification of the important properties of a representation-changing algorithm and then use this specification to implement the algorithm. We illustrate the use of this approach by applying it to the development of two novel representation-changers.

The presentation of the results is organized in five sections. In Section 2, we give two examples of simple domains where representation changes drastically improve the efficiency of problem solving. In Section 3, we describe a novel representation-changing algorithm, which improves the efficiency of problem solving by removing unnecessary operators from the description of a problem domain. In Section 4, we use this algorithm to illustrate our general model of representation-changing algorithms and systematic approach to designing such algorithms. In Section 5, we apply the systematic approach to the design of another novel representation-changing algorithm, which improves the effectiveness of the ALPINE abstraction-generator [Knoblock, 1994] by replacing some predicates in the domain description with more specific predicates. Finally, we summarize the results in Section 6.

## 2 Examples of improving representation

For every problem-solving system and almost every problem, even a simple one, we can find a representation that makes the problem very hard or even unsolvable. These hard representations may not be cumbersome or artificial: a natural representation of a problem is often inappropriate for problem solving. Finding a good representation may be a complex creative task that involves extensive search.

We present two examples of situations where changing the representation is essential for efficient performance of the PRODIGY problem-solver. We will describe algorithms for automatically making these changes in Sections 3 and 5.

**Example 1: Three-Rocket Transportation.**
Consider a planning domain with a planet, three moons, three rockets, and several boxes [Stone and Veloso, 1994]. Initially, all boxes and all three rockets are on the planet. A rocket can carry any number of boxes to any moon. After a rocket has arrived to its destination, it *cannot* be refueled and used again; thus, each rocket can be launched only once. The task of a problem-solving system is to find a plan for delivering certain boxes to certain moons. (We do not care about the resulting locations of specific rockets, as long as every box has reached its proper destination.) For example, if we want to send `box-1` and `box-2` to `moon-1`, and `box-3` to `moon-3`, we can achieve the goal with the following plan:

Load `box-1` and `box-2` into `rocket-1`,
and `box-3` into `rocket-2`.

Send rocket-1 to moon-1 and rocket-2 to moon-3.
Unload both rockets.

To describe a current state of the domain, we have to specify the locations of each rocket and each box, which can be done with three predicates: (at <rocket> <place>), (at <box> <place>), and (in <box> <rocket>), where the "<..>" brackets denote variables; for example, <rocket> is a variable that denotes an arbitrary rocket. We obtain literals describing a state of the domain by substituting specific constants for variables. For example, the literal (at rocket-1 planet) means that rocket-1 is on the planet and (in box-1 rocket-1) means that box-1 is inside rocket-1. The basic operations in the domain, called *operators*, are described by their *preconditions* and *effects*. All preconditions of an operator must hold before the execution of the operator (that is, the preconditions are conjunctive); the effects are the results of the execution.

The user may describe the operations in the Three-Rocket domain by the three operators shown in Figure 1(a). This description, however, makes the problem very hard for PRODIGY: the system tries to use the same rocket for transporting boxes to different moons. PRODIGY performs a long search to discover that each rocket can fly to only one moon [Stone and Veloso, 1994]. If we increase the number of moons and rockets, the problem-solving time grows exponentially. We can use control rules to reduce the search, but the Three-Rocket domain requires a complex set of rules, which are difficult to hand-code or learn automatically.

We can, however, improve the efficiency of PRODIGY by replacing the fly operator with three more specific operators, shown in Figure 1(b). These new operators encode explicitly the knowledge that each rocket flies to only one moon. The use of these operators allows PRODIGY to solve three-rocket transportation problems almost without search. We thus have removed some actions from the domain, leaving a subset of actions sufficient for solving all transportation problems.

The new domain description works only for three rockets and three moons, whereas the original description worked for any number of rockets and moons. Thus, we have improved efficiency by limiting the set of problems that we can solve.

**Example 2: Tower of Hanoi.**
We next show an example of a representation change that enables the ALPINE learning algorithm [Knoblock, 1994] to generate an abstraction hierarchy, which reduces the search complexity of PRODIGY.

We consider the Tower-of-Hanoi puzzle with three disks (Figure 2). The user may describe the states of this puzzle with a single predicate, (on <disk> <peg>), and the operations with the operator shown in Figure 2(a). Most problem-solvers find the solution to the puzzle by an extensive search; the search time grows exponentially with the length of a solution plan. The search could be reduced by using an abstraction hierarchy of predicates [Knoblock, 1994], but the problem description in Figure 2(a) does not allow us to generate a hierarchy of predicates, since this description contains only one predicate.

We may remedy the situation by replacing the predicate (on <disk> <peg>) with three more specific predicates, (on small <peg>), (on medium <peg>), and (on large <peg>), which specify, respectively, the positions of the small, medium, and large disk. Given these new predicates, the ALPINE algorithm generates the three-level abstraction hierarchy shown in Figure 2(b), which reduces the complexity of PRODIGY's search from exponential to linear.

We have again improved efficiency by limiting the set of problems that we can solve: the new description works only for three disks, whereas the old one worked for any number of disks.

## 3 Example of a representation-changer

We describe a novel representation-changing algorithm, called *Operator-Remover*, which detects operators that can be removed from a problem domain without impairing our ability to solve problems. The algorithm removes these unnecessary operators, thus reducing the branching factor of search by a problem-solver and, therefore, improving the efficiency of the problem-solver.

The *Operator-Remover* algorithm is able to automatically perform the representation change in the Three-Rocket domain, described in Section 2: the algorithm determines that all transportation problems can be solved by sending rocket-1 to moon-1, rocket-2 to moon-2, and rocket-3 to moon-3, and removes the operators for the other flights from the domain description. This representation change significantly reduces the complexity of search by the PRODIGY problem-solver.

The main problem in designing an algorithm for removing unnecessary operators is to ensure that the reduced set of operators, generated by the algorithm, allows us to find near-optimal (satisficing) solutions to most problems in the domain. An improper selection of operators may result in generating non-optimal solutions.

To illustrate a possible loss of an optimal solution, let us suppose that the cost of loading or unloading a box in the Three-Rocket domain is 1, the cost of launching rocket-1 or rocket-2 is 2, and the cost of launching rocket-3 is 4. The goal of sending box-1 to moon-3 may be achieved by the plan "(load box-1 rocket-1), (fly rocket-1 moon-3), (unload box-1 rocket-1)," the total cost of which is 4. However, the reduced set of operators shown in Figure 1(b) forces a problem-solver to use rocket-3 for sending a box to moon-3. If the problem-solver uses this reduced set of operators, it finds the plan

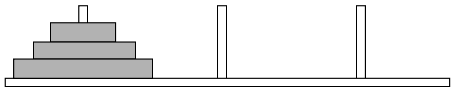
```
Operator LOAD                 || Operator UNLOAD              || Operator FLY
Parameters:                   || Parameters:                  || Parameters:
  <box> <rocket> <place>      ||   <box> <rocket> <place>     ||   <rocket> <moon>
Preconds:                     || Preconds:                    || Preconds:
  (at <box> <place>)          ||   (in <box> <rocket>)        ||   (at <rocket> planet)
  (at <rocket> <place>)       ||   (at <rocket> <place>)      || Effects:
Effects:                      || Effects:                     ||   Del: (at <rocket> planet)
  Del: (at <box> <place>)     ||   Del: (in <box> <rocket>)   ||   Add: (at <rocket> <moon>)
  Add: (in <box> <rocket>)    ||   Add: (at <box> <place>)    ||
```

(a) Initial representation of the operations in the domain.

```
Operator                       || Operator                       || Operator
  FLY-ROCKET-1-TO-MOON-1       ||   FLY-ROCKET-2-TO-MOON-2       ||   FLY-ROCKET-3-TO-MOON-3
Preconds:                      || Preconds:                      || Preconds:
  (at rocket-1 planet)         ||   (at rocket-2 planet)         ||   (at rocket-3 planet)
Effects:                       || Effects:                       || Effects:
  Del: (at rocket-1 planet)    ||   Del: (at rocket-2 planet)    ||   Del: (at rocket-3 planet)
  Add: (at rocket-1 moon-1)    ||   Add: (at rocket-2 moon-2)    ||   Add: (at rocket-3 moon-3)
```

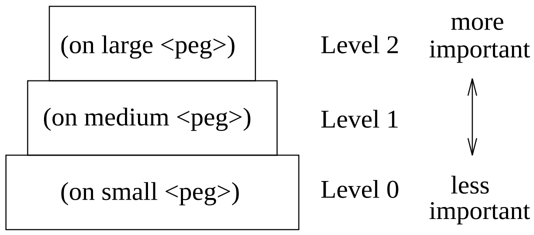
(b) Search-saving representation of the `fly` operation.

Figure 1: Changing the representation of the Three-Rocket domain to reduce PRODIGY's search: problem solving with the initial representation requires an extensive search, whereas the new representation of the `fly` operation enables PRODIGY to solve the problem with little search.



```
Operator MOVE
Parameters: <disk> <from-peg> <to-peg>
Preconds:
  forall <other-disk>
    if (smaller <other-disk> <disk>)
      then ((not (on <other-disk> <from-peg>))
        and (not (on <other-disk> <to-peg>)))
Effects:
  Del: (on <disk> <from-peg>)
  Add: (on <disk> <to-peg>)
```

(a) Representation of the operations in the puzzle.




(b) Abstraction hierarchy generated by ALPINE.

Figure 2: Changing the representation of the Tower-of-Hanoi puzzle to generate an abstraction hierarchy: the ALPINE algorithm fails to generate a hierarchy for the initial representation, but it generates a three-level hierarchy if we use more specific predicates.

"`(load box-1 rocket-3)`, `(fly-rocket3-to-moon-3)`, `(unload box-1 rocket-3)`," with cost 6. In this example, the ratio of the cost of the solution based on the reduced set of operators and the cost of the optimal solution is $6/4 = 1.5$. This ratio is called the *cost increase* of problem solving with the reduced set of operators.

To address the problem of preserving near-optimal solutions, we use a learning algorithm that (1) tests whether the reduced set of operators preserves the completeness of problem solving and (2) estimates the maximal cost increase. The algorithm generates solutions to example problems based on the reduced set of operators and compares them with optimal solutions. If some problem cannot be solved with the reduced set of operators or the cost increase is larger than a certain constant $C$, the algorithm concludes that the reduced set of operators is not sufficient and adds more operators from the initial domain description to this set. Informally, we can describe this algorithm as follows:

1. Generate an initial reduced set of operators.
2. Ask the user to specify the maximal cost increase, $C$.
3. Repeat "several" times:
   (a) Generate an example problem.
   (b) Find an optimal solution,
       using all operators in the problem domain.
   (c) Try to find a solution
       with cost at most $C \cdot \mathit{Cost}(\mathit{Optimal\text{-}Solution})$,
       using only operators from the reduced set.
   (d) If such a solution is not found,
       then add a new operator to the reduced set.

To complete this algorithm, we must provide procedures for generating an initial reduced set of operators and for selecting a new operator when the current reduced set of operators is not sufficient for problem solving. We have discussed some methods for accomplishing these tasks in [Fink and Yang, 1994].

The input to *Operator-Remover* must include the set of operators in a domain. We may also provide the information about possible initial and goal states of problems, which will make the algorithm more effective. For example, *Operator-Remover* simplifies the description of the Three-Rocket domain only if we specify that the positions of rockets are never a part of the goal. (If we do not specify restrictions on goals, the algorithm assumes by default that all states can be goals.) This additional information about possible problems is called an *optional* input.

We next show the upper bound on the probability that the use of the reduced set of operators results in the loss of completeness or in finding too costly solutions. The derivation of this bound may be found in [Fink, 1995].

Suppose that we use the problem-solving algorithm with the reduced set of operators generated by *Operator-Remover*. Let $\delta$ denote the probability that, given a randomly selected solvable problem, the algorithm fails to find a solution the cost of which is at most $C$ times larger than the cost of the optimal solution. In other words, $\delta$ is the probability that the problem-solver does not find a solution or finds a solution that is too costly. Let $m$ denote the number of example problems considered by the *Operator-Remover* algorithm and $n$ denote the number of operators in the problem domain. If $m > n^2$, then the "failure" probability $\delta$ is bounded by the following inequality:

$$\delta \leq n^2/m.$$

This relationship between the number of examples and the failure probability is called the *sample complexity* of the learning algorithm. The inequality shows that the failure probability can be made arbitrarily small by increasing the number of training examples.

## 4 Specification of representation-changers

We have described the *Operator-Remover* algorithm, which removes operators from a domain description in order to reduce the branching factor of search while preserving completeness and near-optimality of problem solving. We say that removing operators is the *type* of the representation change performed by *Operator-Remover* and that reducing the branching factor while preserving completeness and near-optimality is the *purpose* of the representation change.

When implementing a representation-changing algorithm, we must decide on the *type and purpose* of the representation change performed by the algorithm. Restricting representation change to a specific type limits the space of alternative representations explored by the algorithm, whereas specifying the purpose shows exactly in which way the algorithm must improve the representation. We must also determine which other algorithms it calls as subroutines and identify the features of a problem domain that must be included into the algorithm's input.

These top-level decisions form a *specification* of a representation-changer. Specifications allow us to (1) formally describe the desired properties of a representation-changing algorithm before implementing the algorithm and (2) separate these properties from specific methods used in the implementation. In Section 5, we will illustrate the use of specifications in designing a representation-changer.

When developing a specification of a representation-changing algorithm, we must make sure that the specification describes a useful representation change and that we can implement a simple and efficient algorithm that satisfies the specification. We compose specifications from the following five parts:

**Type of representation change.** When designing a representation-changer, we first select a type of representation change, such as removing operators (as in *Operator-Remover*) and replacing predicates with more

**Generating an abstraction hierarchy:** Decomposing the set of predicates in the domain description into several subsets, according to the "importance" of predicates [Sacerdoti, 1974].

**Replacing operators with macros:** Replacing some operators in the domain description with macros constructed from these operators [Fikes *et al.*, 1972].

**Selecting primary effects of operators:** Choosing certain "important" effects of operators and using operators only for achieving their important effects [Fink and Yang, 1993].

**Removing operators:** Deleting unnecessary operators from the domain description (see Example 1 in Section 2).

**Generating more specific predicates:** Replacing a predicate in the domain description with several more specific predicates, which together describe the same set of ground literals (see Example 2 in Section 2).

Figure 3: Examples of types of representation changes for improving the efficiency of problem solving.

specific ones (as in the Tower-of-Hanoi example in Section 2). In Figure 3, we give some examples of types of representation changes for improving the efficiency of general-purpose problem-solving systems.

**Purpose of representation change.** A representation changing algorithm should find a representation that satisfies the following three requirements:

- *Efficiency*: Problem solving with the new representation is efficient for most of the frequently encountered problems. We use an objective function to estimate the efficiency of problem solving with a new representation.

- *Near-Completeness*: Most solvable problems remain solvable in the new representation. We measure this factor by the percentage of solvable problems that become unsolvable after the representation change.

- *Near-Optimality*: The change of representation preserves optimal or near-optimal solutions to most problems and the problem-solving algorithm is able to find near-optimal solutions. We define the optimal solution as the solution with the lowest total cost of operators. We measure the optimality factor by the largest increase in the cost of solutions generated by the algorithm.

For example, in the *Operator-Remover* algorithm we use the number of operators in the reduced set as an objective function for estimating the problem-solving efficiency: the fewer operators, the more efficient problem solving. The percentage of problems that may become unsolvable is bounded by $n^2/m$, where $n$ is the number of operators in the domain and $m$ is the number of example problems considered by *Operator-Remover* (see Section 3). Finally, the increase in the cost of solutions is bounded by the user-specified maximal cost increase, $C$.

*To summarize, we view the purpose of a representation change as maximizing a specific objective function, while preserving near-completeness and near-optimality, which is the central idea of the approach. This formal specification of the purpose shows exactly in which way we improve the representation and enables us to evaluate the results of a representation change.*

**The use of other algorithms.** We specify here the problem-solving and learning algorithms that the representation-changer uses as subroutines and the purpose of using these algorithms. For example, *Operator-Remover* calls the PRODIGY problem-solver to solve the example problems used in learning.

**Required input.** We list here the parts of the domain description that we must include into the representation-changer's input. For example, the required input to *Operator-Remover* includes the description of operators in the domain.

**Optional input.** Finally, we specify the additional information about the domain that may be used by the representation-changer. If the user inputs this information, the representation-changer uses it to generate a better representation; in the absence of such information, the algorithm uses some default values. The optional input may include useful knowledge about the properties of the domain, such as control rules and a list of primary effects of operators, and restrictions on the types of problems that we will solve with a new representation. For example, we may specify restrictions on the initial and goal states of problems as an optional input to *Operator-Remover*. If we do not provide this information, the algorithm assumes by default that problems may have any initial and goal states.

We summarize the specification of the *Operator-Remover* algorithm in Figure 4.

## 5 Designing a representation-changer

We have described a method for specifying the important properties of representation-changing algorithms, which enables us to abstract these important properties from implementational details. We now illustrate the application of this method to the design of a novel representation-changer, called *Instantiator*, which improves the effectiveness of the ALPINE abstraction-generator by replacing some predicates in the domain description with more specific predicates. *Instantiator* is able to perform the representation change in the Tower-of-Hanoi domain, described in Section 2.

The ALPINE abstraction-generator is a very fast, *Type of representation change:* Removing operators.

*Purpose of representation change:* Minimizing the number of remaining operators, while ensuring that the cost increase is within the user-specified bound $C$.

*The use of other algorithms:* A problem-solver, used in checking completeness and estimating the cost increase of problem solving with the reduced set of operators.

*Required input:* Description of the operators; maximal cost increase.

*Optional input:* Restrictions on the possible initial and goal states.

Figure 4: Specification of the *Operator-Remover* representation changing algorithm.

polynomial-time algorithm that generates an abstraction hierarchy of predicates using several restrictions on the importance levels of predicates [Knoblock, 1994]. The restrictions are based on the relations between preconditions and effects of operators. A level of the abstraction hierarchy generated by ALPINE consists of predicates of equal "importance."

If the user describes a problem domain by a small number of general predicates, the ALPINE algorithm usually fails to generate an abstraction hierarchy, since the algorithm cannot distribute special cases of a general predicate between several levels of abstraction. We encountered this problem in the Tower-of-Hanoi domain described by a single predicate, `(on <disk> <peg>)` (see Figure 2). ALPINE cannot distribute this single predicate between several levels of abstraction and, thus, it cannot generate a multi-level abstraction hierarchy.

We may avoid the problem of too general predicates by instantiating all variables in the domain description, thus replacing each general predicate with a large number of fully instantiated predicates and each operator with a number of fully instantiated operators. The full instantiation, however, may lead to a combinatorial explosion in the number of instantiated predicates and operators.

Knoblock proposed a more sophisticated technique for replacing general predicates with specific ones, in which the user divides the values of variables into classes [Knoblock, 1991]. For example, in the Tower-of-Hanoi domain, the user may indicate that the disks are divided into three classes, `Small`, `Medium`, and `Large`, whereas all pegs are of the same class. ALPINE uses this information to generate three specific predicates, `on-small`, `on-medium`, and `on-large`, and then generates more specific operators by incorporating these more specific predicates into the operator description. This approach, however, requires the user's help. Besides, a large number of classes in a complex domain may still lead to a combinatorial explosion.

To avoid the explosion, the algorithm must automatically identify the variables whose instantiation leads to increasing the number of abstraction levels, and instantiate only these variables. For example, in the Tower-of-Hanoi domain, the algorithm must instantiate `<disk>` and leave `<peg>` uninstantiated.

To perform the instantiation, the algorithm must know all possible values of the variables. For example, it must know that the possible values of `<disk>` are `small`, `medium`, and `large`. The algorithm should also use predicates that describe unchangeable relations between the values of variables, such as `(smaller small medium)`, which constrain possible instantiations of variables in the description of operators. Using these constraints, the algorithm generates fewer instantiated operators, which enables ALPINE to use less restrictions in generating an abstraction hierarchy and, thus, produce a hierarchy with more levels.

We can now express the requirements to an algorithm for generating more specific predicates as a specification of a representation-changer; we show the specification in Figure 5.

We next design a greedy algorithm, called *Instantiator*, that satisfies this specification. The algorithm tries to instantiate different variables, one by one, and calls ALPINE to determine whether instantiating a variable leads to an increase in the number of abstraction levels. If the instantiation does not increase the number of levels, the algorithm leaves the variable uninstantiated and tries to instantiate another variable. Informally, we can describe the *Instantiator* algorithm as follows:

For every variable $v$ in the domain description:
1. For every operator $Op$ that contains $v$,
   instantiate $v$ in $Op$ with all its possible values,
   thus replacing $Op$ by set of more specific operators.
2. Call ALPINE to generate an abstraction hierarchy
   for the resulting domain description.
3. If the number of levels in this new hierarchy is larger
   than that in the hierarchy before instantiating $v$,
   then leave $v$ instantiated in the domain description,
   else go back to the description with uninstantiated $v$.

For example, suppose that we apply the *Instantiator* algorithm to the Tower-of-Hanoi domain described with a single predicate, `(on <disk> <peg>)`. The algorithm will notice that instantiating the variable `<disk>` results in generating a three-level abstraction hierarchy, whereas instantiating the variable `<peg>` does not increase the number of abstraction levels. The algorithm will thus instantiate `<disk>` and generate the three-level hierarchy shown in Figure 2(b).

Note that the *Instantiator* algorithm does not try to instantiate several variables at once and, therefore, it fails to find a good instantiation if the only way to increase the number of abstraction levels is to instantiate several variables together. We may avoid this problem

*Type of representation change:* Generating more specific predicates.

*Purpose of representation change:* Maximizing the number of levels in the abstraction hierarchy generated by ALPINE.

*The use of other algorithms:* ALPINE, used in generating an abstraction hierarchy based on more specific predicates.

*Required input:* Description of the operators; the possible values of the variables in the domain description.

*Optional input:* Predicates describing unchangeable relations between the values of variables.

Figure 5: Specification of the *Instantiator* algorithm.

by modifying the algorithm in such a way that it considers instantiations of couples and triples of variables when instantiating single variables does not improve the abstraction hierarchy.

## 6 Conclusions

We have described a systematic approach to the analysis and design of representation-changing algorithms, based on the formal specification of the important properties of these algorithms. We have demonstrated that the use of specifications simplifies the task of designing representation-changers and evaluating their performance. When developing a representation-changer, we first formalize its desirable properties and then implement a learning or search algorithm with these properties.

We have presented two novel representation-changing algorithms, *Operator-Remover* and *Instantiator*. In [Fink, 1995], we have described the use of specifications in designing two other, more complex representation-changers, *Prim-Tweak* [Fink and Yang, 1993] and *Margie* [Fink and Yang, 1992], which improve the efficiency of problem solving by selecting primary effects of operators. We have demonstrated experimentally that *Prim-Tweak* and *Margie* considerably improve the performance in a variety of planning domains [Fink and Yang, 1995].

We plan to extend and formalize the structure and language of specifications, study methods for determining which specifications describe useful representation changes, develop techniques for implementing representation-changing algorithms according to specifications, and demonstrate the practical usefulness of representation-changers by applying them to large-scale domains. We also plan to study algorithms capable of improving languages for problem description.


## Acknowledgements

I gratefully acknowledge the valuable help of Herbert Simon, Jaime Carbonell, and Manuela Veloso, who guided me in my research and writing. I am grateful to Nevin Heintze, who encouraged me to write this paper. I also owe thanks to Henry Rowley, Yury Smirnov, and Alicia Pérez for their comments and suggestions.


## References


[Allen *et al.*, 1992] John Allen, Pat Langley, and Stan Matwin. Knowledge and regularity in planning. In *Proceedings of the AAAI 1992 Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse*, pages 7–12, 1992.

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence 3*, pages 131–171. American Elsevier Publishers, 1968.

[Bacchus and Yang, 1994] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43–100, 1994.

[Carbonell, 1990] Jaime G. Carbonell, editor. *Machine Learning: Paradigms and Methods*. MIT Press, Boston, MA, 1990.

[Fikes *et al.*, 1972] Richard E. Fikes, P. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4), 1972.

[Fink and Yang, 1992] Eugene Fink and Qiang Yang. Automatically abstracting effects of operators. In *Proceedings of the First International Conference on AI Planning Systems*, pages 243–251, 1992.

[Fink and Yang, 1993] Eugene Fink and Qiang Yang. Characterizing and automatically finding primary effects in planning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1374–1379, 1993.

[Fink and Yang, 1994] Eugene Fink and Qiang Yang. Automatically selecting and using primary effects in planning: Theory and experiments. Technical Report CMU-CS-94-206, School of Computer Science, Carnegie Mellon University, 1994.

[Fink and Yang, 1995] Eugene Fink and Qiang Yang. Planning with primary effects: Experiments and analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1995.

[Fink, 1995] Eugene Fink. Systematic approach to the design of representation-changing algorithms. Technical Report CMU-CS-95-120, School of Computer Science, Carnegie Mellon University, 1995.

[Hibler, 1994] David Hibler. Implicit abstraction by thought experiments. In *Proceedings of the Workshop on Theory Reformulation and Abstraction*, pages 9–26, 1994.

[Kaplan and Simon, 1990] Craig A. Kaplan and Herbert A. Simon. In search of insight. *Cognitive Psychology*, 22:374–419, 1990.

[Knoblock, 1991] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-120.

[Knoblock, 1994] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68, 1994.

[Korf, 1980] Richard E. Korf. Toward a model of representation changes. *Artificial Intelligence*, 14:41–78, 1980.

[Korf, 1985] Richard E. Korf. Macro-operators: A week method for learning. *Artificial Intelligence*, 25:35–77, 1985.

[Laird *et al.*, 1987] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Larkin *et al.*, 1988] Jill H. Larkin, Frederick Reif, Jaime G. Carbonell, and Angela Gugliotta. FERMI: A flexible expert reasoner with multi-domain inferencing. *Cognitive Psychology*, 12:101–138, 1988.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1988. Technical Report CMU-CS-88-133.

[Mooney, 1988] Raymond J. Mooney. Generalizing the order of operators in macro-operators. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 270–283, San Mateo, CA, 1988. Morgan Kaufmann.

[Newell and Simon, 1972] Allen Newell and Herbert A. Simon. *Human Problem Solving*. Prentice Hall, Englewood Cliffs, NJ, 1972.

[Newell *et al.*, 1960] Allen Newell, J. C. Shaw, and Herbert A. Simon. A variety of intelligent learning in a general problem solver. In Marshall C. Yovits, editor, *International Tracts in Computer Science and Technology and Their Applications*, volume 2: Self-Organizing Systems, pages 153–189. Pergamon Press, New York, NY, 1960.

[Newell, 1965] Allen Newell. Limitations of the current stock of ideas about problem solving. In A. Kent and O. Tualbee, editors, *Electronic Information Handling*. Spartan Books, Washington, DC, 1965.

[Newell, 1966] Allen Newell. On the representations of problems. In *Computer Science Research Reviews*. Carnegie Institute of Technology, Pittsburgh, PA, 1966.

[Newell, 1992] Allen Newell. Unified theories of cognition and the role of Soar. In J. A. Michon and A. Akyürek, editors, *Soar: A Cognitive Architecture in Perspective*, pages 25–79. Kluwer Academic Publishers, Netherlands, 1992.

[Penberthy and Weld, 1992] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial-order planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation in Reasoning*, pages 103–114, 1992.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Shell and Carbonell, 1989] Peter Shell and Jaime G. Carbonell. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[Simon *et al.*, 1985] Herbert A. Simon, K. Kotovsky, and J. R. Hayes. Why are some problems hard? Evidence from the Tower of Hanoi. *Cognitive Psychology*, 17:248–294, 1985.

[Stone and Veloso, 1994] Peter Stone and Manuela M. Veloso. Learning to solve complex planning problems: Finding useful auxiliary problems. In *Proceedings of the AAAI 1994 Fall Symposium on Planning and Learning*, pages 137–141, 1994.

[Stone *et al.*, 1994] Peter Stone, Manuela M. Veloso, and Jim Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 164–169, 1994.

[Veloso *et al.*, 1995] Manuela M. Veloso, Jaime G. Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.