Eugene Fink

Changes of Problem Representation: Theory and Experiments

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Contents

Part I

Introduction

1. Motivation

> Could you restate the problem? Could you restate it still differently?
> — George Polya [1957], *How to Solve It.*

The performance of all reasoning systems crucially depends on problem representation: the same problem may be easy or difficult, depending on the way we describe it. Researchers in psychology, cognitive science, and artificial intelligence have accumulated much evidence on the importance of appropriate representations for human problem solvers and AI systems.

In particular, psychologists have found out that human subjects often simplify difficult problems by changing their representation. The ability to find an appropriate problem reformulation is a crucial skill for mathematicians, physicists, economists, and experts in many other areas. AI researchers have shown the impact of changes in problem description on the performance of search systems and pointed out the need to automate problem reformulation.

Although researchers have long realized the importance of effective representations, they have done little investigation in this area, and the notion of "good" representations has remained at an informal level. The user has traditionally been responsible for providing appropriate problem descriptions, as well as for selecting search algorithms that effectively use these descriptions.

The purpose of our work is to automate the process of revising problem representation in AI systems. We formalize the concept of representation, explore its role in problem solving, and develop a system that evaluates and improves representations in the PRODIGY problem-solving architecture.

The work on the system for changing representations has consisted of two main stages, described in Parts II and III. First, we outline a framework for the development of algorithms that improve problem descriptions and apply it to designing several novel algorithms. Second, we construct an integrated AI system, which utilizes the available description improvers and problem solvers. The system is named $\mathcal{S}$HAPER for its ability to change the shape of problems and their search spaces. We did not plan this name to be an acronym; however, it may be retroactively deciphered as *Synergy of Hierarchical Abstraction, Primary Effects, and other Representations.* The central component of the $\mathcal{S}$HAPER system is a control module that selects appropriate algorithms for each given problem.

We begin by explaining the concept of representations in problem solving (Section 1.1), illustrating their impact on problem complexity (Section 1.2), and reviewing the previous work on representation changes (Section 1.3). We then outline our approach to the automation of representation improvements (Section 1.4) and give a summary of the main results (Section 1.5).

1.1 Representations in problem solving

Informally, a *problem representation* is a certain view of a problem and an approach to solving it. Scientists have considered various formalizations of this concept; its exact meaning varies across research contexts. The representation of a problem in an AI system may include the initial encoding of the problem, data structures for storing relevant information, production rules for drawing inferences about the problem, and heuristics that guide the search for a solution.

We explain the meaning of representation in our research and introduce related terminology. First, we give informal examples that illustrate this notion (Section 1.1.1). Second, we review several alternative formalizations (Section 1.1.2) and define the main notions used in the work on the $\mathcal{S}$HAPER system (Section 1.1.3). Third, we discuss the role of representation in problem solving (Section 1.1.4).

1.1.1 Informal examples

We consider two examples that illustrate the use of multiple representations. In Section 1.2, we will give a more technical example, which involves representation changes in the PRODIGY architecture.

Representations in geometry. Mathematicians have long mastered the art of constructing and fine-tuning sophisticated representations, which is one of their main tools for addressing complex research tasks [Polya, 1957]. For example, when a scientist works on a difficult geometry problem, she usually tries multiple approaches: pictorial reasoning, analytical techniques, trigonometric derivations, and computer simulations.

These approaches differ not only in the problem encoding, but also in the related mental structures and strategies [Qin and Simon, 1992]. For example, the mental techniques for analyzing geometric sketches are different from the methods for solving trigonometric equations.

A mathematician may attempt many alternative representations of a given problem, moving back and forth among promising approaches [Kaplan and Simon, 1990]. For instance, she may consider several pictorial representations, then try analytical techniques, and then abandon her analytical model and go back to one of the pictures.

If several different representations provide useful information, the mathematician may use them in parallel and combine the resulting inferences. This synergetic use of alternative representations is a standard mathematical technique. In particular, proofs of geometric results often include equations along with pictorial arguments.

The search for an appropriate representation is based on two main processes: the retrieval or construction of candidate representations and the evaluation of their utility. The first process may involve look-up of a matching representation in a library of available strategies, modification of an "almost" matching representation, or development of a completely new approach. For example, the mathematician may reuse an old sketch, draw a new one, or even devise a new framework for solving this type of problem.

After constructing a new representation, the mathematician estimates its usefulness for solving the problem. If it does not look promising, she may prune it right away or store it as a back-up alternative; for example, she may discard the sketches that clearly do not help. Otherwise, she uses the representation and evaluates the usefulness of the resulting inferences.

To summarize, different representations of a given problem support different inference techniques, and the choice among them determines the effectiveness of the problem-solving process. Construction of an appropriate representation can be a difficult task, and it can require a search in a certain space of alternative representations.

Driving directions. We next give an example of representation changes in everyday life and show that the choice of representation can be important even for simple tasks. In this example, we consider the use of directions for driving to an unfamiliar place.

Most drivers employ several standard techniques for describing a route, such as a sketch of the streets that form the route, pencil marks on a city map, and verbal directions for reaching the destination. When a driver chooses one of these techniques, she commits to certain mental structures and strategies. For instance, if the driver uses a map, she has to process pictorial information and match it to the real world. On the other hand, the execution of verbal instructions requires discipline in following the described steps and attention to the relevant landmarks.

When the driver selects a representation, she should consider her goals, the effectiveness of alternative representations for achieving these goals, and the related trade-offs. For instance, she may describe the destination by its address, which is a convenient way for recording it in a notebook or quickly communicating to others; however, the address alone may not be sufficient for finding the place without a map. The use of accurate verbal directions is probably the most convenient way for driving to the destination. On the other hand, a map may help to identify gas stations or restaurants close to the route; moreover, it can be a valuable tool if the driver gets lost.

A representation...

- includes a machine language for the description of reasoning tasks and a specific encoding of a given problem in this language [Amarel, 1968].
- is the space expanded by a solver algorithm during its search for a solution [Newell and Simon, 1972].
- is the state space of a given problem, formed by all legal states of the simulated world and transitions between them [Korf, 1980].
- "consists of both data structures and programs operating on them to make new inferences" [Larkin and Simon, 1987].
- determines a mapping from the behavior of an AI system on a certain set of inputs to the behavior of another system, which performs the same task on a similar set of inputs [Holte, 1988].

Fig. 1.1. Definitions of representation in artificial intelligence and cognitive science. These definitions are not equivalent, and thus they lead to different formal models.

If an appropriate representation is not available, the driver may construct it from other representations. For instance, if she has the destination address, she may find a route on a map and write down directions. When people consider these representation changes, they often weigh the expected simplification of the task against the cost of performing the changes. For instance, even if the driver believes that written directions facilitate the trip, she may decide that they are not worth the writing effort.

To summarize, this example shows that people employ multiple representations not only for complex problems, but also for routine tasks. When people repeatedly perform a certain task, they develop standard representations and techniques for constructing them. Moreover, the familiarity with the task facilitates the selection among available representations.

1.1.2 Alternative definitions of representation

Although AI researchers agree on their intuitive understanding of representation, they have not yet developed a standard formalization of this notion. We review several formal models and discuss their similarities and differences; in Figure 1.1, we summarize the main definitions.

Problem formulation. Amarel [1961; 1965; 1968] was first to point out the impact of representation on the efficiency of search algorithms. He considered some problems of reasoning about actions in a simulated world and discussed their alternative formulations in the input language of a search algorithm. The discussion included two types of representation changes: modifying the encoding of a problem and translating it to different languages.

In particular, he demonstrated that a specific formulation of a problem determines its *state space,* that is, the space of possible states of the simulated world and transitions between them. Amarel pointed out that the efficiency of problem solving depends on the size of the state space, as well as on the

allowed transitions, and that change of a description language may help to reveal hidden properties of the simulated world.

Van Baalen [1989] adopted a similar view in his doctoral work on a theory of representation design. He defined a representation as a mapping from concepts to their syntactic description in a formal language and implemented a program that automatically improved descriptions of simple reasoning tasks.

Problem space. Newell and Simon [1972] investigated the role of representation in human problem solving. In particular, they observed that the human subject always encodes a given task in a *problem space,* that is, "some kind of space that represents the initial situation presented to him, the desired goal situation, various intermediate states, imagined or experienced, as well as any concepts he uses to describe these situations to himself" [Newell and Simon, 1972].

They defined a representation as the subject's problem space, which determines partial solutions considered by the human solver during his search for a complete solution. This definition is also applicable to AI systems since all problem-solving algorithms are based on the same principle of searching among partial solutions.

Observe that the problem space may differ from the state space of the simulated world. In particular, the subject may disregard some of the allowed transitions and, on the other hand, consider impossible world states. For instance, when people work on difficult versions of the Tower of Hanoi, they sometimes attempt illegal moves [Simon *et al.*, 1985]. Moreover, the problem solver may abstract from the search among world states and use an alternative view of partial solutions. In particular, the search algorithms in the PRODIGY architecture explore the space of transition sequences (see Section 2.2), which is different from the space of world states.

State space. Korf [1980] described a formal framework for changing representations and used it in designing a system for automatic improvement of the initial representation. He developed a language for describing search problems and defined a representation as a specific encoding of a given problem in this language. The encoding includes the initial state of the simulated world and operations for transforming the state; hence, it defines the state space of the problem.

Korf pointed out the correspondence between the problem encoding and the resulting state space, which allowed him to view a representation as a space of named states and transitions between them. This view underlied his techniques for changing representation. In particular, he defined a representation change as a transformation of the state space and considered two main types of transformations: *isomorphism* and *homomorphism.* An isomorphic representation change was renaming of the states without changing the structure of the space. On the other hand, a homomorphic transformation was a reduction of the space by abstracting some states and transitions.

Observe that Korf's notion of representation did not include the behavior of a problem-solving algorithm. Since performance depended not only on the state space but also on the search strategies, a representation in his model did not uniquely determine the efficiency of problem solving.

Data and programs. Simon suggested a general definition of representation as "data structures and programs operating on them," and used it in the analysis of reasoning with pictorial representations [Larkin and Simon, 1987]. When describing the behavior of human solvers, he viewed their initial encoding of a given problem as a "data structure," and the available productions for modifying it as "programs." Since the problem encoding and rules for changing it determined the subject's search space, this view was similar to the earlier definition by Newell and Simon [1972].

If we apply Simon's definition in other research contexts, the notions of data structures and programs may take different meanings. The general concept of "data structures" encompasses any form of a system's input and internal representation of related information. Similarly, the term "programs" may refer to any strategies and procedures for processing a given problem.

In particular, when considering an AI architecture with several search engines, we may view the available engines as "programs" and the information passed among them as "data structures." We will use this approach to formalize representation changes in the PRODIGY system.

System's behavior. Holte [1988] developed a framework for the analysis and comparison of learning systems, which included rigorous mathematical definitions of task domains and their representations. He considered representations of domains rather than specific problems, which distinguished his view from the earlier definitions.

A domain in Holte's framework includes a set of elementary entities, a collection of primitive functions that describe the relationships among entities, and legal compositions of primitive functions. For example, we may view the world states as elementary objects and transitions between them as primitive functions. A domain specification may include not only a description of reasoning tasks, but also a behavior of an AI system on these tasks.

A representation is a mapping between two domains that encode the same reasoning tasks. This mapping may describe a system's behavior on two different encodings of a problem. Alternatively, it may show the correspondence between the behavior of two different systems that perform the same task.

1.1.3 Representations in the *SHAPER* system

The previous definitions of representation have been aimed at the analysis of its role in problem solving, but researchers have not applied theoretical results to automation of representation changes. Korf utilized his formal model in the development of a general-purpose system for improving representations, but

A **problem solver** is an algorithm that performs some type of reasoning task. When we invoke this algorithm, it inputs a given problem and searches for a solution; it may solve the problem or report a failure.

A **problem description** is an input to a problem solver. In most systems, it includes allowed operations, available objects, the initial state of the world, a logical statement describing the goals, and possibly some heuristics for guiding the search.

A **domain description** is the part of the problem description that is common for a certain class of problems. It usually does not include specific objects, an initial state, or a goal.

A **representation** is a domain description with a problem solver that uses this description. A representation change may involve improving a description, selecting a new solver, or both.

A **description changer** is an algorithm for improving domain descriptions. When we invoke the changer, it inputs a given domain and modifies its description.

A **system for changing representations** is an AI system that automatically improves domain descriptions and matches them with appropriate problem solvers.

Fig. 1.2. Definitions of the main objects in the SHAPER system. These notions underlie our formal model of automatic representation changes.

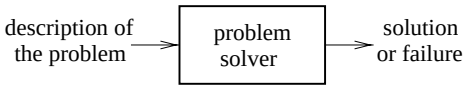
he encountered several practical shortcomings of the model, which prevented complete automation of search for appropriate representations.

Since the main purpose of our work is the construction of a fully automated system, we develop a different formal model, which facilitates the work on SHAPER. We follow Simon's view of representation as "data structures and programs operating on them;" however, the notion of data structures and programs in SHAPER differs from their definition in the research on human problem solving [Newell and Simon, 1972]. We summarize our terminology in Figure 1.2.

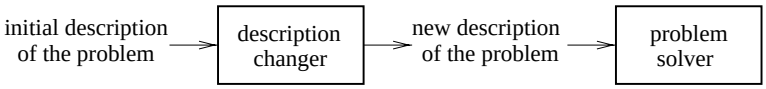
The SHAPER system uses PRODIGY search algorithms, which play the role of "programs" in Simon's definition. We illustrate the functions of a solver algorithm in Figure 1.3(a): given a problem, the algorithm searches for its solution and either finds some solution or terminates with a failure. In Chapter 6, we will discuss two types of failures: exhausting the available search space and reaching a time limit.

A *problem description* is an input to the solver algorithm, which encodes a certain reasoning task. This notion is analogous to Amarel's "problem formulation," which is a part of his definition of representation. The solver's input must satisfy certain syntactic and semantic rules, which form the *input language* of the algorithm.

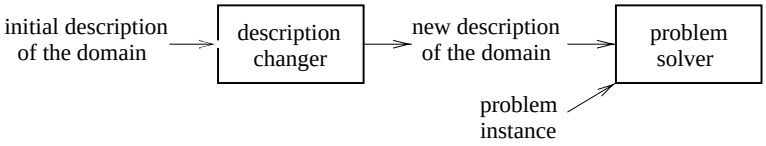
When the initial description of a problem does not obey these rules, we have to translate it into the input language before applying the solver [Paige and Simon, 1966; Hayes and Simon, 1974]. If a description satisfies the language rules but causes a long search, we may modify it for efficiency reasons.



(a) Use of a problem-solving algorithm.



(b) Changing the problem description before application of a problem solver.



(c) Changing the domain description.

Fig. 1.3. Description changes in problem solving. A changer algorithm generates a new description, and then a solver algorithm uses it to search for a solution.

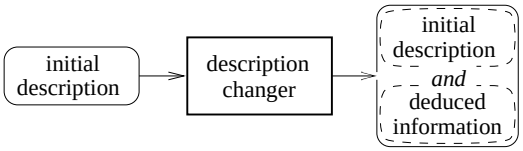
A *description-changing algorithm* is a procedure for converting the initial problem description into an input to a problem solver, as illustrated in Figure 1.3(b). The conversion may serve two goals: (1) translating the problem into the input language and (2) improving performance of the solver.

The $\mathcal{S}$HAPER system performs only the second type of description changes. In Figure 1.4, we show the three main categories of these changes: decomposing the initial problem into smaller subproblems, enhancing the description by adding relevant information, and replacing the original problem encoding with a more appropriate encoding.

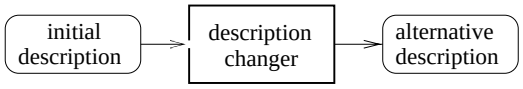
Note that there are no clear-cut boundaries between these categories. For example, suppose that we apply some *abstraction procedure* (see Section 1.2) to determine the relative importance of different problem features and then use important features in constructing an outline of a solution. We may view it as enhancement of the initial description with the estimates of importance. Alternatively, we may classify abstraction as decomposition of the original problem into two subproblems: constructing a solution outline and then turning it into a complete solution.

A problem description in PRODIGY consists of two main parts, a *domain description* and a *problem instance.* The first part includes the properties of a simulated world, which is called the *problem domain.* For example, if we apply PRODIGY to solve the Tower-of-Hanoi puzzle (see Section 1.2.1), the domain description specifies the legal moves in this puzzle. The second part

(a) Decomposing a problem into subproblems. We may simplify a reasoning task by breaking it into smaller subtasks. For example, a driver may subdivide the search for an unfamiliar place into two stages: getting to the appropriate highway exit and finding her way from the exit. In Section 1.2.2, we will give a technical example of problem decomposition based on an abstraction hierarchy.



(b) Enhancing a problem description. If some important information is not explicit in the initial description, we may deduce it and add to the description. If the addition of new information affects the problem-solving behavior, we view it as a description change. For instance, a mathematician may enhance a geometry sketch by an auxiliary construction. As another example, we may improve performance of PRODIGY by adding control rules.



(c) Replacing a problem description. If the initial description contains unnecessary data, then improvements may include not only additional relevant information, but also the deletion of irrelevant data. For example, a mathematician may simplify her sketch by erasing some lines. In Section 5.2, we will describe a technique for detecting irrelevant features of PRODIGY problems.

Fig. 1.4. Main categories of description changes. We may (**a**) subdivide a problem into smaller tasks, (**b**) extend the initial description with additional knowledge, and (**c**) replace a given problem encoding with a more effective encoding.

encodes a particular reasoning task, which includes an initial state of the simulated world and a goal specification. For example, a problem instance in the Tower-of-Hanoi Domain consists of the initial positions of all disks and their desired final positions.

The PRODIGY system first parses the domain description and converts it into internal structures that encode the simulated world. Then, PRODIGY uses this internal encoding in processing specific problem instances. Observe that the input description determines the system's internal encoding of the

domain. Thus, the role of domain descriptions in our model is similar to that of "data structures" in the general definition.

When using a description changer, we usually apply it to the domain encoding and utilize the resulting new encoding for solving multiple problem instances (Figure 1.3c). This strategy allows us to amortize the running time of the changer algorithm over several problems.

A *representation* in SHAPER consists of a domain description and a problem-solving algorithm that operates on this description. Observe that, if the algorithm does not make random choices, the representation uniquely defines the search space for every problem instance. This observation relates our definition to Newell and Simon's view of representation as a search space.

We use this definition in the work on a *representation-changing system,* which automates the two main tasks involved in improving representations. First, it analyzes and modifies the initial domain description with the purpose of improving the search efficiency. Second, it selects an appropriate solver algorithm for the modified domain description.

1.1.4 The role of representation

Researchers have used several different frameworks for defining the concept of representation. Despite these differences, most investigators have reached consensus on their main qualitative conclusions:

- The choice of a representation affects the complexity of a given problem; both human subjects and AI systems are sensitive to changes in the problem representation.
- Finding the right approach to a given problem is often a difficult task, which may require a heuristic search in a space of alternative representations.
- Human experts employ advanced techniques for construction and evaluation of new representations, whereas amateurs often try to utilize the original problem description.

Alternative representations differ in explicit information about properties of the problem domain. Every representation hides some features of the domain and highlights other features [Newell, 1965; Van Baalen, 1989; Peterson, 1994]. For example, when a mathematician describes a geometric object by a set of equations, she hides visual features of the object and highlights some of its analytical properties.

Explicit representation of important information enhances the performance. For instance, if a student of mathematics cannot solve a problem, the teacher may help her by pointing out the relevant features of the task [Polya, 1957]. As another example, we may improve efficiency of an AI system by encoding useful information in control rules [Minton, 1988], macro operators [Fikes *et al.*, 1972], or an abstraction hierarchy [Sacerdoti, 1974].

On the other hand, the explicit representation of irrelevant data may have a negative effect. In particular, when a mathematician tries to utilize

some seemingly relevant properties of a problem, she may attempt a wrong approach. If we provide irrelevant information to an AI system and do not mark this information as unimportant, then the system attempts to use it, which takes extra computation and often leads to exploring useless branches of the search space. For example, if we allow the use of unnecessary extra operations, the branching factor of search increases, resulting in a larger search time [Stone and Veloso, 1994].

Since problem-solving algorithms differ in their use of available information, they perform efficiently with different domain descriptions. Moreover, the utility of explicit knowledge about the domain may depend on a specific problem instance. We usually cannot find a "universal" description, which works well for all solver algorithms and problem instances. The task of constructing good descriptions has traditionally been left to the user.

The relative performance of solver algorithms also depends on specific problems. Most analytical and experimental studies have shown that different search techniques are effective for different classes of problems, and no solver algorithm can consistently outperform all its competitors [Minton *et al.*, 1994; Stone *et al.*, 1994; Knoblock and Yang, 1994; Knoblock and Yang, 1995; Smirnov, 1997]. To ensure efficiency, the user has to make an appropriate selection among the available algorithms.

To address the representation problem, researchers have designed a number of algorithms that deduce hidden properties of a given domain and improve the domain description. For example, they constructed systems for learning control rules [Mitchell *et al.*, 1983; Minton, 1988], replacing operators with macros [Fikes *et al.*, 1972; Korf, 1985a], abstracting unimportant features of a domain [Sacerdoti, 1974; Knoblock, 1993], and reusing past problem-solving episodes [Hall, 1989; Veloso, 1994].

These algorithms are themselves sensitive to changes in problem encoding, and their ability to learn useful information depends on the initial description. For instance, most systems for learning control rules require a certain generality of predicates in the domain encoding, and they become ineffective if predicates are either too specific or too general [Etzioni and Minton, 1992; Veloso and Borrajo, 1994].

As another example, abstraction algorithms are very sensitive to the description of available operators [Knoblock, 1993]. If the operator encoding is too general, or the domain includes unnecessary operations, these algorithms fail to construct an abstraction hierarchy. In Section 1.2, we will illustrate such failures and discuss related description improvements.

To ensure the effectiveness of learning algorithms, the user has to decide which algorithms are appropriate for a given domain. She may also need to adjust the domain description for the selected algorithms. An important next step in AI research is to develop a system that automatically accomplishes these tasks.

1.2 Examples of representation changes

All AI systems are sensitive to the description of input problems. If we use an inappropriate domain encoding, then even simple problems may become difficult or unsolvable. Researchers have noticed that novices often construct ineffective domain descriptions because intuitively appealing encodings are often inappropriate for AI problem solving.

On the other hand, expert users prove proficient in finding good descriptions; however, the construction of a proper domain encoding is often a difficult task, which requires creativity and experimentation. The user usually begins with a semi-effective description and tunes it, based on the results of problem solving. If the user does not provide a good domain encoding, then automatic improvements are essential for efficiency.

To illustrate the need for description changes, we present a puzzle domain, whose standard encoding is inappropriate for PRODIGY (Section 1.2.1). We then show modifications to the initial encoding that drastically improve efficiency (Sections 1.2.1–1.2.4), and discuss the choice of an appropriate problem solver (Section 1.2.5). The $\mathcal{S}$HAPER system is able to perform these improvements automatically.

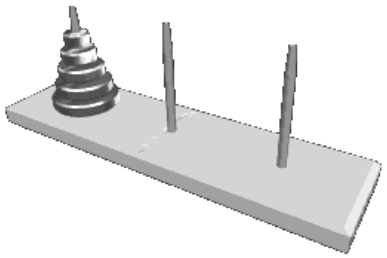
1.2.1 Tower-of-Hanoi Domain

We consider the Tower-of-Hanoi puzzle, shown in Figure 1.5, which has proved difficult for most problem-solving algorithms, as well as for human subjects. It once served as a standard test for AI systems, but it gradually acquired the negative connotation of a "toy" domain. We utilize this puzzle to illustrate basic description changes in $\mathcal{S}$HAPER; however, we will use larger domains for the empirical evaluation of the system.

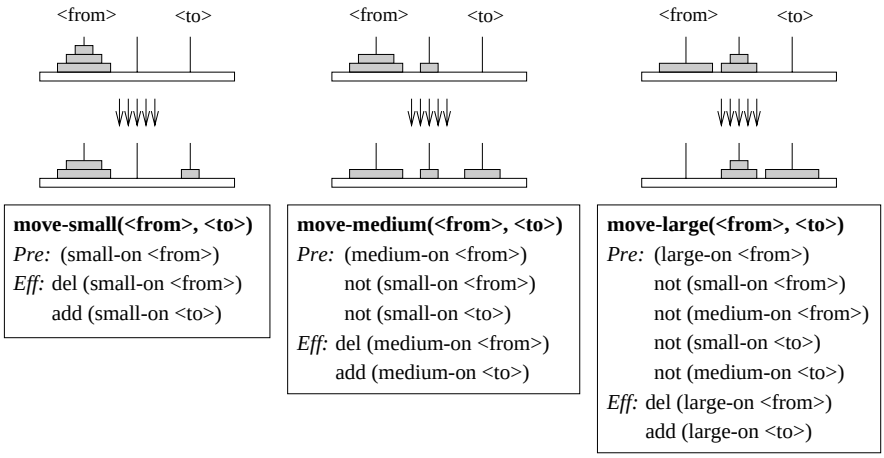
The puzzle consists of three vertical pegs and several disks of different sizes. Every disk has a hole in the middle, and several disks may be stacked on a peg (Figure 1.5a). The rules allow moving disks from peg to peg, one disk at a time; however, the rules do not allow placing any disk above a smaller one. In Figure 1.7, we show the state space of the three-disk puzzle.

When using a classical AI system, we have to specify predicates for encoding of world states. If the Tower-of-Hanoi puzzle has three disks, we may describe its states with three predicates, which denote the positions of the disks: (small-on <peg>), (medium-on <peg>), and (large-on <peg>), where <peg> is a variable that denotes an arbitrary peg. We obtain literals describing a specific state by substituting the appropriate constants in place of variables. For instance, the literal (small-on peg-1) means that the small disk is on the first peg.

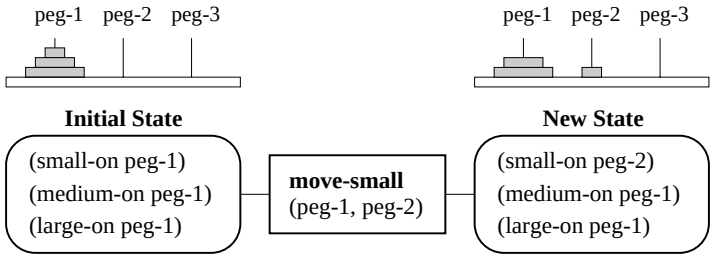
The legal moves are encoded by production rules for modifying the world state, which are called *operators.* The description of an operator consists of *precondition predicates,* which must hold before its execution, and *effects,* which specify the predicates added to the world state or deleted from the state upon the execution. In Figure 1.5(b), we give an encoding of all allowed moves



(a) Tower-of-Hanoi puzzle.



(b) Encoding of operations in the three-disk puzzle.



(c) Example of executing an instantiated operator.

Fig. 1.5. Tower-of-Hanoi Domain and its encoding in PRODIGY. The player may move disks from peg to peg, one at a time, without ever placing a disk on top of a smaller one. The traditional task is to move all disks from the left-hand peg to the right-hand (Figure 1.6).

in the three-disk puzzle. This encoding is based on the PRODIGY domain language, described in Sections 2.2 and 2.3; however, we slightly deviate from the exact PRODIGY syntax in order to improve readability.

The <from> and <to> variables in the operator description denote arbitrary pegs. When a problem-solving algorithm uses an operator, it instantiates the variables with specific constants. For example, if the algorithm needs to move the small disk from peg-1 to peg-2, then it can execute the operator **move-small**(peg-1,peg-2) (Figure 1.5c). The precondition of this operator is (small-on peg-1); that is, the small disk must initially be on peg-1. The execution results in deleting (small-on peg-1) from the current state and adding (small-on peg-2).

In Figure 1.6, we show the encoding of a classic problem in the Tower-of-Hanoi Domain, which requires moving all three disks from peg-1 to peg-3, and give the shortest solution to this problem. The initial state of the problem corresponds to the left corner of the state-space triangle in Figure 1.7, whereas the goal state is the right corner.

1.2.2 Constructing an abstraction hierarchy

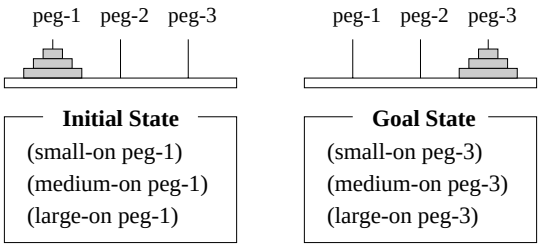
Most AI systems solve a given problem by exploring the space of partial solutions rather than expanding the problem's state space. That is, the nodes in their search space represent incomplete solutions, which may not correspond to paths in the state space.

This strategy allows efficient reasoning in large-scale domains, which have intractable state spaces; however, it causes a major inefficiency in the Tower-of-Hanoi Domain. For example, if we apply PRODIGY to the problem in Figure 1.6, the system considers more than one hundred thousand partial solution, and the search takes ten minutes on a Sun Sparc 5 computer.

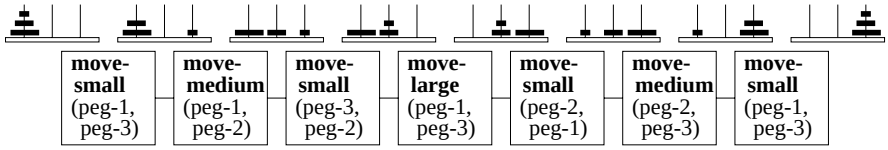
We may significantly improve the performance by using an *abstraction hierarchy* [Sacerdoti, 1974], which enables the system to subdivide problems into simpler subproblems. To construct a hierarchy, we assign different levels of importance to predicates in the domain encoding. In Figure 1.8(a), we give the standard hierarchy for the three-disk Tower of Hanoi.

The system first constructs an abstract solution at level 2 of the hierarchy, ignoring the positions of the small and medium disks. We show the state space of the abstracted puzzle in Figure 1.8(b) and its solution in Figure 1.8(c). Then, PRODIGY steps down to the next lower level and inserts operators for moving the medium disk. At this level, the system cannot add new **move-large** operators, which limits its state space. We give the level-1 space in Figure 1.8(d) and the corresponding solution in Figure 1.8(e). Finally, the system shifts to the lowest level and inserts **move-small** operators, thus constructing the complete solution (Figures 1.8f and 1.8g).

In Table 1.1, we give the running times for solving six Tower-of-Hanoi problems, without and with abstraction. We have obtained these results using



(a) Encoding of the problem.



(b) Shortest solution.

Fig. 1.6. Example of a problem in the Tower-of-Hanoi Domain. We need to move all three disks from peg-1 to peg-3. The shortest solution contains seven steps.

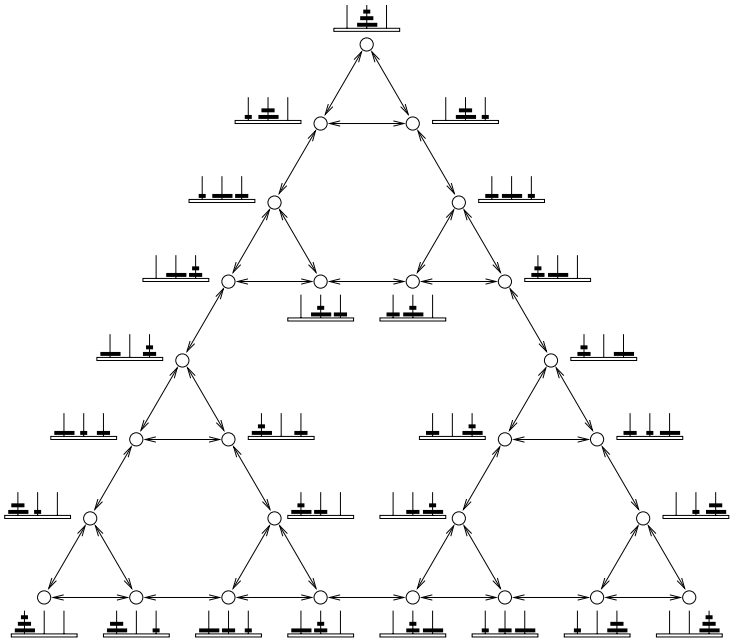
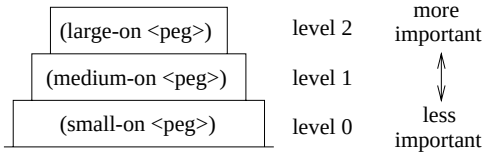
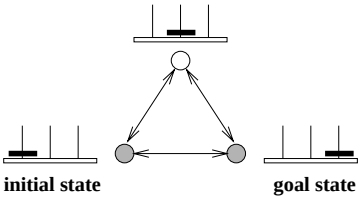


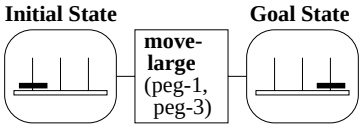
Fig. 1.7. State space of the three-disk Tower of Hanoi. We illustrate all possible configurations of the puzzle (circles) and legal transitions between them (arrows). The initial state of the problem in Figure 1.6 is the left corner of the triangle, and the goal is the right corner.



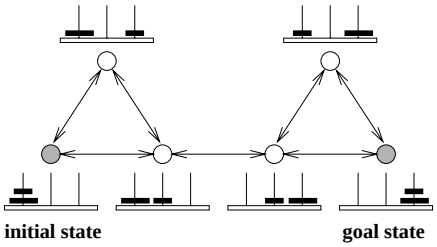
(a) Abstraction hierarchy of predicates.



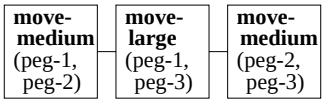
(b) State space at level 2.



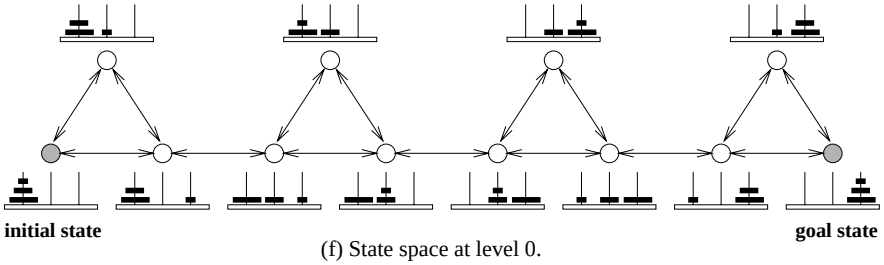
(c) Solution at level 2.



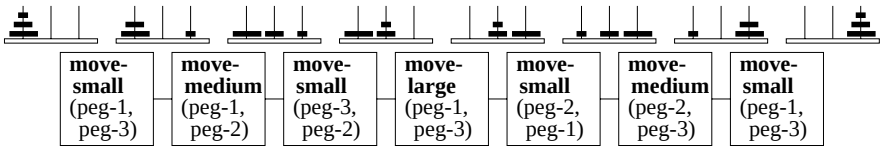
(d) State space at level 1.



(e) Solution at level 1.



(f) State space at level 0.



(g) Solution at level 0.

Fig. 1.8. Abstraction problem solving in the Tower-of-Hanoi Domain with a three-level hierarchy (**a**). First, the system disregards the small and medium disks, thus solving the simplified one-disk puzzle (**b, c**). Then, it inserts the missing movements of the medium disk (**d, e**). Finally, it steps down to the lowest level of abstraction and adds **move-small** operators (**f, g**).

Table 1.1. PRODIGY performance on six Tower-of-Hanoi problems. We give running times in seconds for problem solving without and with the abstraction hierarchy.

	number of a problem						mean
	1	2	3	4	5	6	time
without abstraction	2.0	34.1	275.4	346.3	522.4	597.4	296.3
using abstraction	0.5	0.4	1.9	0.3	0.5	2.3	1.0

Table 1.2. PRODIGY performance in the Extended Tower-of-Hanoi Domain, which allows two-disk moves. We give running times in seconds for three different domain descriptions: without primary effects, with primary effects, and using primary effects along with abstraction. The results show that primary effects not only reduce the search, but also allow the construction of an effective abstraction hierarchy.

	number of a problem						mean
	1	2	3	4	5	6	time
without prim. effects	85.3	1.1	505.0	>1800.0	31.0	172.4	>432.4
using prim. effects	0.5	1.2	16.6	144.8	77.5	362.5	100.5
prims and abstraction	0.5	0.3	2.5	0.2	0.2	3.1	1.1

a Lisp implementation of PRODIGY on a Sun Sparc 5 machine; they show that abstraction drastically reduces search.

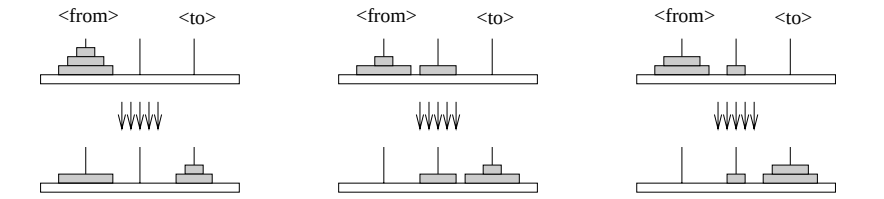
Knoblock [1993] has investigated abstraction problem solving in PRODIGY and developed the ALPINE system, which automatically assigns importance levels to predicates. We have extended Knoblock's technique and implemented the *Abstractor* algorithm, which serves as one of the description changers in SHAPER.

1.2.3 Selecting primary effects

Suppose that we deviate from the standard rules of the Tower-of-Hanoi puzzle and allow moving two disks together (Figure 1.9a). The new operators enable us to construct shorter solutions for most problems. For example, we can move all three disks from `peg-1` to `peg-3` in three steps (Figure 1.9b).

This change in the rules simplifies the puzzle for humans, but it makes most problems harder for PRODIGY. The extra operations lead to a higher branching factor, thus increasing the search space. Moreover, *Abstractor* fails to generate a hierarchy for the domain with two-disk moves. In the first row of Table 1.2, we give the results of using the extended set of operators to solve the six sample problems. For every problem, we set a 1800-second limit, and the system ran out of time on problem 4.

To reduce the branching factor, we may select *primary effects* of some operators and force the system to use these operators only for achieving their primary effects. For example, we may indicate that the main effect of **move-sml-mdm** is the new position of the medium disk. That is, if the system's only goal is moving the small disk, then it must not consider this operator. Note that an inappropriate choice of primary effects may compro-

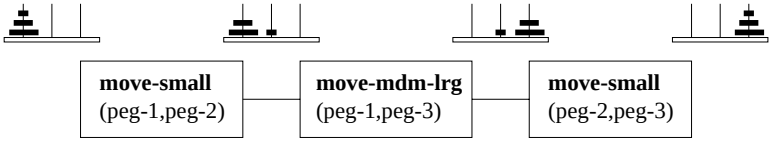


move-sml-mdm
(<from>, <to>)
Pre: (small-on <from>)
(medium-on <from>)
Eff: del (small-on <from>)
add (small-on <to>)
del (medium-on <from>)
add (medium-on <to>)

move-sml-lrg
(<from>, <to>)
Pre: (small-on <from>)
(large-on <from>)
not (medium-on <from>)
not (medium-on <to>)
Eff: del (small-on <from>)
add (small-on <to>)
del (large-on <from>)
add (large-on <to>)

move-mdm-lrg
(<from>, <to>)
Pre: (medium-on <from>)
(large-on <from>)
not (small-on <from>)
not (small-on <to>)
Eff: del (medium-on <from>)
add (medium-on <to>)
del (large-on <from>)
add (large-on <to>)

(a) Encoding of two-disk moves.



(b) Solution to the example problem.

operators	primary effects
move-sml-mdm (<from>,<to>)	del (medium-on <from>) add (medium-on <to>)
move-sml-lrg (<from>,<to>)	del (large-on <from>) add (large-on <to>)
move-mdm-lrg (<from>,<to>)	del (large-on <from>) add (large-on <to>)

(c) Selection of primary effects.

Fig. 1.9. Extension to the Tower-of-Hanoi Domain, which includes operators for moving two disks at a time, and primary effects of these additional operators.

mise completeness, that is, make some problems unsolvable. We will describe techniques for ensuring completeness in Section 3.2.

In Figure 1.9(c), we list the primary effects of the two-disk moves. The use of the selected primary effects improves the system's performance on most sample problems (see the middle row of Table 1.2). Furthermore, it enables *Abstractor* to build the three-level hierarchy, which reduces search by two orders of magnitude (see the last row).

The SHAPER system includes an algorithm for selecting primary effects, called *Margie*[1], which automatically performs this description change. The *Margie* algorithm is integrated with *Abstractor*: it chooses primary effects with the purpose of improving abstraction.

1.2.4 Partially instantiating operators

The main drawback of the *Abstractor* algorithm is its sensitivity to syntactic features of a domain encoding. In particular, if predicates in the domain description are too general, *Abstractor* may fail to construct a hierarchy.

For instance, suppose that the user replaces the predicates (small-on <peg>), (medium-on <peg>), and (large-on <peg>) with a more general predicate (on <disk> <peg>), which allows greater flexibility in the encoding of operators and problems (Figure 1.10a). In particular, it enables the user to utilize universal quantifications (Figure 1.10b).

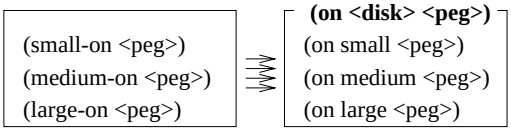
Since the resulting description contains only one predicate, the abstraction algorithm cannot generate a hierarchy. To remedy this problem, we can construct partial instantiations of (on <disk> <peg>) and apply *Abstractor* to build a hierarchy of these instantiations (Figure 1.10c). We have implemented a description-changing algorithm, called *Refiner*, that generates partially instantiated predicates for improving the effectiveness of *Abstractor*.

1.2.5 Choosing a problem solver

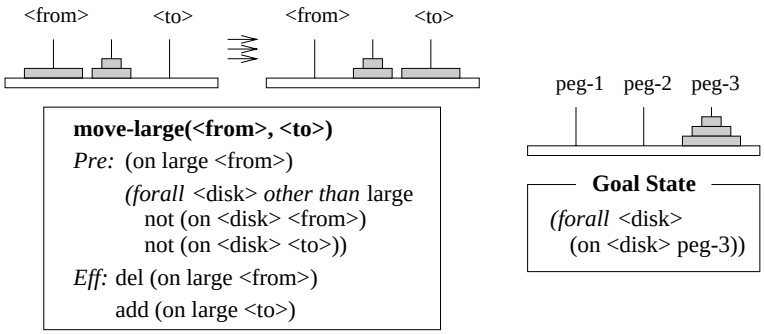
The efficiency of search depends not only on the domain description, but also on the choice of a solver algorithm. To illustrate the importance of a solver, we consider the application of two different search strategies, called SAVTA and SABA, to problems in the Extended Tower-of-Hanoi Domain.

Veloso and Stone [1995] developed these strategies for guiding PRODIGY search (see Section 2.2.5). Experiments have shown that the relative performance of SAVTA and SABA varies across domains, and the choice between them may be essential for efficient problem solving. We consider two modes

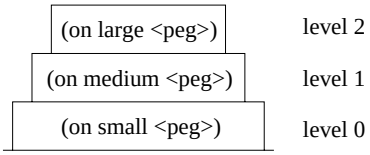
[1] The *Margie* procedure (pronounced *mär'gē*) is named after my friend, Margie Roxborough. Margie and her husband John invited me to stay at their place during the Ninth CSCSI Conference in Vancouver, where I gave a talk on related results. This algorithm is not related to the MARGIE system (*mär'jē*) for parsing and paraphrasing simple stories in English, implemented by Schank *et al.* [1975].



(a) Replacing the predicates with a more general one.



(b) Examples of using the general predicate to encode operators and problems.



(c) Hierarchy of partially instantiated predicates.

Fig. 1.10. General predicate (on <disk> <peg>) in the encoding of the Tower of Hanoi. This predicate enables the user to utilize quantifications in describing operators and goals, but it causes a failure of the *Abstractor* algorithm. The system has to generate partial instantiations of (on <disk> <peg>) before invoking *Abstractor*.

for using each strategy: with a bound on the search depth and without limiting the depth. A depth bound helps to prevent an extensive exploration of inappropriate branches in the search space; however, it also results in pruning some solutions from the search space. Moreover, if a bound is too tight, it may lead to pruning all solutions, thus compromising the system's completeness.

In Table 1.3, we give the results of applying SAVTA and SABA to six Tower-of-Hanoi problems. These results suggest that SAVTA without a time limit is more effective than the other three techniques; however, the evidence is not statistically significant. In Section 7.4, we will describe a method for estimating the probability that a selected problem-solving technique is the best among the available techniques. We can apply this method to determine the chances that SAVTA without a time limit is indeed the most effective among the four techniques; it gives the probability estimate of 0.47.

Table 1.3. Performance of SAVTA and SABA in the Extended Tower-of-Hanoi Domain with primary effects and abstraction. We give running times in seconds for search with and without a depth bound. The data suggest that SAVTA without a time limit is the most efficient among the four search techniques, but this conclusion is not statistically significant.

	number of a problem						mean
	1	2	3	4	5	6	time
SAVTA							
with depth bound	0.51	0.27	2.47	0.23	0.21	3.07	1.13
w/o depth bound	0.37	0.28	0.61	0.22	0.21	0.39	0.35
SABA							
with depth bound	5.37	0.26	>1800.00	2.75	0.19	>1800.00	>601.43
w/o depth bound	0.45	0.31	1.22	0.34	0.23	0.51	0.51

If we apply $\mathcal{S}$HAPER to many Tower-of-Hanoi problems, it can accumulate more data on the performance of the candidate strategies before adopting one of them. The system's control module combines exploitation of the past performance information with collecting additional data. First, $\mathcal{S}$HAPER applies heuristics for rejecting inappropriate search techniques; then, the system experiments with promising search algorithms until it accumulates enough data for identifying the most effective algorithm.

Note that we have not accounted for solution quality in evaluating PRODIGY performance in the Tower-of-Hanoi Domain. If the user is interested in near-optimal solutions, then $\mathcal{S}$HAPER has to analyze the trade-off between running time and solution quality, which may result in selecting different domain descriptions and search strategies. For instance, a depth bound reduces the length of generated solutions, which may be a fair payment for the increase in search time. As another example, search without an abstraction hierarchy usually yields better solutions than abstraction problem solving. In Chapter 7, we will describe a statistical technique for evaluating trade-offs between speed and quality.

1.3 Related work

We review past results on representation changes, including psychological experiments (Section 1.3.1), AI techniques for reasoning with multiple representations (Sections 1.3.2 and 1.3.3), and theoretical frameworks (Section 1.3.4).

We will later describe some other directions of past research, related to specific aspects of the reported work. In particular, we will give the history of the PRODIGY architecture in Section 2.1.1, outline the previous research on abstraction problem solving in Section 4.1.1, and review the work on automatic evaluation of problem solvers in Section 7.1.1.

1.3.1 Psychological evidence

The choice of an appropriate representation is one of the main themes of Polya's famous book *How to Solve It.* Polya showed that the selection of an effective approach to a problem is a crucial skill for a student of mathematics. Gestalt psychologists also paid particular attention to reformulation of problems [Duncker, 1945; Ohlsson, 1984].

Explorations in cognitive science have yielded much evidence that confirms Polya's pioneering insight. Researchers have shown that description changes affect the problem difficulty and that the performance of human experts depends on their ability to find a representation that fits a given task [Gentner and Stevens, 1983; Simon, 1989].

Newell and Simon [1972] studied the role of representation during their investigation of human problem solving. They observed that human subjects always construct some representation of a given problem: "Initially, when a problem is first presented, it must be recognized and understood. Then, a problem space must be constructed or, if one already exists in LTM, it must be evoked. Problem spaces can be changed or modified during the course of problem solving" [Newell and Simon, 1972].

Simon [1979; 1989] continued the investigation of representations in human problem solving and studied their role in a variety of cognitive tasks. In particular, he tested the utility of different mental models in the Tower-of-Hanoi puzzle. Simon [1975] noticed that most subjects gradually improved their mental representations in the process of solving the puzzle.

Hayes and Simon [1974; 1976; 1977] investigated the effect of isomorphic changes in task description. Specifically, they analyzed difficult isomorphs of the Tower of Hanoi [Simon *et al.*, 1985] and found out that "changing the written problem instructions, without disturbing the isomorphism between problem forms, can affect by a factor of two the times required by subjects to solve a problem" [Simon, 1979].

Larkin and Simon [1981; 1987] explored the role of multiple mental models in solving physics and math problems, with a particular emphasis on pictorial representations. They observed that mental models of skilled scientists differ from those of novices, and expertly constructed models are crucial for solving scientific problems. Qin and Simon [1992] came to a similar conclusion during their experiments on the role of imagery in understanding special relativity.

Kook and Novak [1991] also studied alternative representations in physics. They implemented the APEX program, which used multiple representations of physics problems, handled incompletely specified tasks, and performed transformations among several types of representations. Their conclusions about the significance of appropriate representation agreed with Simon's results.

Kaplan and Simon [1990] explored representation changes in solving the Mutilated-Checkerboard problem. They noticed that the participants of their experiments first tried to utilize the initial representation and then searched

for a more effective approach. Kaplan [1989] implemented a production system that simulated the representation shift of successful human subjects.

Tabachneck [1992] studied the utility of pictorial reasoning in economics. She implemented the CaMeRa production system, which modeled human reasoning with multiple representations [Tabachneck-Schijf *et al.*, 1997].

Boehm-Davis *et al.* [1989], Novak [1995], and Jones and Schkade [1995] showed the importance of representations in software development. Their results confirmed that people are sensitive to changes in problem description and that improvement of the initial description is often a difficult task.

The reader may find a detailed review of past results in Peterson's [1996] collection of articles on reasoning with multiple representations. It includes several different views of representation, as well as evidence on the importance of representation in physics, mathematics, economics, and other areas.

1.3.2 Automatic representation changes

AI researchers recognized the significance of representation in the very beginning of their work on automated reasoning. In particular, Amarel [1961; 1968; 1971] discussed the effects of representation on the behavior of search algorithms, using the Missionaries-and-Cannibals problem to illustrate his main points. Newell [1965; 1966] showed that the complexity of some games and puzzles strongly depends on the representation and emphasized that "hard problems are solved by finding new viewpoints; i.e., new problem spaces" [Newell, 1966].

Later, Newell with several other researchers implemented the Soar system [Laird *et al.*, 1987; Tamble *et al.*, 1990; Newell, 1992], which utilized multiple descriptions to facilitate search and learning; however, their system did not generate new representations. The user was still responsible for constructing domain descriptions and providing guidelines for their effective use.

Larkin *et al.* [1988] took a similar approach in their work on the FERMI expert system, which accessed several representations of task-related knowledge and used them in parallel. This system required the user to provide appropriate representations of the input knowledge. The authors of FERMI encoded "different kinds of knowledge at different levels of granularity" and demonstrated that "the principled decomposition of knowledge according to type and level of specificity yields both power and cross-domain generality" [Larkin *et al.*, 1988].

Research on automatic change of domain descriptions has mostly been limited to the design of separate learning algorithms that perform specific types of improvements. Examples of these special-purpose algorithms include systems for replacing operators with macros [Korf, 1985; Mooney, 1988; Cheng and Carbonell, 1986; Shell and Carbonell, 1989], changing the search space by learning heuristics [Newell *et al.*, 1960; Langley, 1983] and control rules [Minton *et al.*, 1989; Etzioni, 1993; Veloso and Borrajo, 1994; Pérez,

1995], generating abstraction hierarchies [Sacerdoti, 1974; Knoblock, 1993], and replacing a given problem with a simplified problem [Hibler, 1994].

The authors of these systems have observed that utility of most learning techniques varies across domains, and their blind application may worsen efficiency in some domains; however, researchers have not automated selection among available learning systems and left it as the user's responsibility.

1.3.3 Integrated systems

The Soar architecture includes a variety of general-purpose and specialized search algorithms, but it does not have a top-level mechanism for selecting an algorithm that fits a given task. The classical problem-solving systems, such as SIPE [Wilkins, 1988], PRODIGY [Carbonell *et al.*, 1990; Veloso *et al.*, 1995], and UCPOP [Penberthy and Weld, 1992; Weld, 1994], have the same limitation: they allow alternative search strategies, but they do not include a central mechanism for selecting among them.

Wilkins and Myers [1995; 1998] have addressed this problem and constructed the Multiagent Planning Architecture, which supports integration of multiple planning and scheduling algorithms. The available search algorithms in their architecture are arranged into groups, called *planning cells.* Every group has a control procedure, called a *cell manager,* which analyzes an input problem and selects an algorithm for solving it. Some cell managers are able to break a given task into subtasks and distribute them among several algorithms.

The architecture includes advanced software tools for incorporating diverse search algorithms with different domain languages; thus, it allows a synergetic use of previously implemented AI systems. Wilkins and Myers have demonstrated the effectiveness of their architecture in constructing centralized problem-solving systems. In particular, they have developed several large-scale systems for air campaign planning.

Since the Multiagent Planning Architecture allows the use of diverse algorithms and domain descriptions, it provides an excellent testbed for the study of search with multiple representations; however, its current capabilities for automated representation changes are very limited.

First, the system has no general-purpose control mechanisms, and the user has to design a specialized manager algorithm for every planning cell. Wilkins and Myers have used fixed selection strategies in the implemented cell managers and have not augmented them with learning capabilities.

Second, the system has no tools for the inclusion of algorithms that improve domain descriptions, and the user must either hand-code all necessary descriptions or incorporate a mechanism for changing descriptions into a cell manager. The authors of the architecture have implemented several specialized techniques for decomposing a problem into subproblems, but have not considered other types of description changes.

Minton [1993a; 1993b; 1996] has investigated the integration of constraint-satisfaction programs and designed the MULTI-TAC system, which combines a number of generic heuristics and search procedures. The system's control module explores the properties of a given domain, selects appropriate search strategies, and combines them into an algorithm for this domain.

The main component of MULTI-TAC's control module is an inductive learning algorithm, which tests the available heuristics on a collection of problems. It guides a beam search in the space of the allowed combinations of heuristics. The system synthesizes efficient constraint-satisfaction algorithms, which perform on par with manually configured programs. The major drawback is significant learning time; when the control module searches for an efficient algorithm, it tests candidate procedures on hundreds of sample problems.

Yang *et al.* [1998] have begun development of an architecture for integration of AI planning techniques. Their architecture, called PLAN++, includes tools for implementing, modifying, and reusing the main elements of planning systems. The purpose is to modularize typical planning algorithms, construct a large library of search modules, and use them as building blocks for new algorithms.

Since the effectiveness of planning techniques varies across domains, the authors of PLAN++ intend to design software tools that enable the user to select appropriate modules and configure them for specific domains. The automation of these tasks is a major open problem, which is closely related to work on representation improvements.

1.3.4 Theoretical results

Researchers have developed theoretical frameworks for some special cases of description changes, including abstraction, replacement of operators with macros, and learning of control rules; however, they have done little study of the common principles underlying different changer algorithms. The results in developing a formal model of representation are also limited, and the general notion of useful representation changes has remained at an informal level.

Most theoretical models are based on the analysis of a search space. When investigating some description change, researchers usually identify its effects on the search space and estimate the resulting reduction in the search time.

In particular, Korf [1985a; 1985b; 1987] investigated the effects of macro operators on the state space and demonstrated that well-chosen macros may exponentially reduce the search time. Etzioni [1992] analyzed the search space of backward-chaining algorithms and showed that an inappropriate choice of macro operators or control rules may worsen the performance. He then derived a condition under which macros reduce the search and compared the utility of macro operators with that of control rules.

Cohen [1992] developed a mathematical framework for the analysis of macro-operator learning, explanation-based generation of control rules, and

chunking. He analyzed mechanisms for reusing solution paths and described a series of learning algorithms that provably improve performance.

Knoblock [1993] explored the benefits and limitations of abstraction, identified conditions that ensure search reduction, and used them in developing an algorithm for automatic abstraction. Bacchus and Yang [1992; 1994] removed some of the assumptions underlying Knoblock's analysis and presented a more general evaluation of abstraction search. They studied the effects of backtracking across abstraction levels, demonstrated that it may impair efficiency, and described a technique for avoiding it.

Giunchiglia and Walsh [1992] proposed a general model of reasoning with abstraction, which captured most of the previous frameworks. They defined an abstraction as a mapping between axiomatic formal systems, studied the properties of this mapping, and classified the main abstraction techniques.

A generalized model for improving representation was suggested by Korf [1980], who formalized representation changes based on the notions of isomorphism and homomorphism of state spaces (see Section 1.1.3). Korf utilized the resulting formalism in his work on automatic representation improvements; however, his model did 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" [Korf, 1980].

1.4 Overview of the approach

The review of previous work has shown that the results are still limited. The main open problems are (1) design of AI systems capable of performing a wide range of representation changes and (2) development of a unified theory of reasoning with multiple representations.

The work on $\mathcal{S}$HAPER is a step toward addressing these two problems. We have developed a framework for evaluating representations and formalized the task of finding an appropriate representation in the space of alternative domain descriptions and solver algorithms. We have applied this framework to design a system that automatically performs several types of representation improvements.

The system includes a collection of problem-solving and description-changing algorithms, as well as a control module that selects and invokes appropriate algorithms (Figure 1.11). The most important result is the control mechanism for intelligent selection among available algorithms and representations. We use it to combine multiple learning and search algorithms into an integrated AI system.

We now explain the main architectural decisions that underlie $\mathcal{S}$HAPER (Section 1.4.1), outline our approach to development of changer algorithms (Section 1.4.2), and briefly describe search in a space of representations (Section 1.4.3).

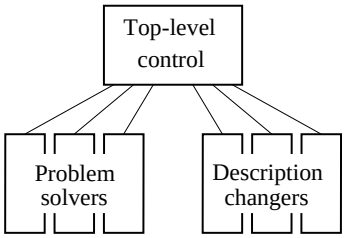


Fig. 1.11. Integration of solver and changer algorithms. SHAPER includes a control module that analyzes a given problem and selects appropriate algorithms.

1.4.1 Architecture of the system

According to our definition, a system for changing representations has to perform two main functions: (1) improvement of a problem description and (2) selection of an algorithm for solving the problem. The key architectural decision underlying SHAPER is the distribution of the first task among multiple changer algorithms. For instance, the description improvement in Section 1.2 has involved three algorithms: *Refiner*, *Margie*, and *Abstractor*.

The central use of separate algorithms differentiates SHAPER from Korf's mechanism for improving representations. It also differs from description-improving cell managers in the Multiagent Planning Architecture.

Every changer algorithm explores a certain space of modified descriptions until finding a new description that improves the system's performance. For example, *Margie* searches among alternative selections of primary effects, whereas *Refiner* explores a space of different partial instantiations.

The control module coordinates the application of description-changing algorithms. It explores a more general space of domain descriptions, using changer algorithms as operators for expanding nodes in this space. The system thus combines the low-level search by changer algorithms with the centralized high-level search. This two-level search prevents a combinatorial explosion in the number of candidate representations, described by Korf [1980].

The other major function of the control module is the selection of problem-solving algorithms for the constructed domain descriptions. To summarize, SHAPER consists of three main parts, illustrated in Figure 1.11:

Library of problem solvers. The system uses search algorithms of the PRODIGY architecture. We have composed the solver library from several different configurations of PRODIGY's search mechanism.

Library of description changers. We have implemented seven algorithms that compose SHAPER's library of changers. They include procedures for selecting primary effects, building abstraction hierarchies, generating partial and full instantiations of operators, and identifying relevant features of the domain.

Efficiency. The primary criterion for evaluating representations in the SHAPER system is the efficiency of problem solving, that is, the average search time.

Near-completeness. A change of representation must not cause a significant violation of completeness; that is, most solvable problems should remain solvable after the change. We measure the completeness violation by the percentage of problems that become unsolvable.

Solution quality. A representation change should not result in significant decline of solution quality. We usually define the quality as the total cost of operators in a solution and evaluate the average increase in solution costs.

Fig. 1.12. Main factors that determine the quality of a representation. We have developed a general utility model that unifies these factors and enables the system to make trade-off decisions.

> **Control module.** The functions of the control mechanism include selection of description changers and problem solvers, evaluation of new representations, and reuse of the previously generated representations. The control module contains statistical procedures for analyzing past performance, heuristics for choosing algorithms in the absence of past data, and tools for manual control.

The control mechanism does not rely on specific properties of solver and changer algorithms, and the user may readily add new algorithms; however, all solvers and changers must use the PRODIGY domain language and access relevant data in the central data structures of the PRODIGY architecture.

We evaluate the utility of representations along three dimensions, summarized in Figure 1.12: the efficiency of search, the number of solved problems, and the quality of the resulting solutions [Cohen, 1995]. The work on changer algorithms involves decisions on trade-offs among these factors. We allow a moderate loss of completeness and solution quality in order to improve efficiency. In Section 6.3, we will describe a general utility function that unifies the three evaluation factors.

When evaluating the utility of a new representation, we have to account for the overall time for improving a domain description, selecting a problem solver, and using the resulting representation to solve given problems. The system is effective only if this overall time is smaller than the time for solving the problems with the initial description and some fixed solver algorithm.

1.4.2 Specifications of description changers

The current version of SHAPER includes seven description changers. We have already mentioned three of them: an extended version of the ALPINE abstraction generator, called *Abstractor*; the *Margie* algorithm, which selects primary effects; and the *Refiner* procedure, which generates partial instantiations of operators.

Selecting primary effects of operators: Choosing "important" effects and using operators only for achieving their important effects (Sections 3.4.1 and 3.5).

Generating an abstraction hierarchy: Decomposing the set of predicates in a domain into subsets, according to their "importance" (Sections 4.1, 4.2, and 5.1).

Generating more specific operators: Replacing an operator with several more specific operators, which together describe the same actions. We generate specific operators by instantiating variables in an operator description (Section 3.4.2).

Generating more specific predicates: Replacing a predicate with more specific predicates, which together describe the same set of literals (Section 4.3).

Identifying relevant features (literals): Determining which features of a domain are relevant to the current task and ignoring the other features (Section 5.2).

Fig. 1.13. Types of description changes in the current version of SHAPER.

Every changer algorithm in SHAPER performs a specific *type* of description change and serves a certain *purpose.* For example, *Margie* selects primary effects of operators, with the purpose of increasing the number of levels in the abstraction hierarchy.

When implementing a changer algorithm, we must decide on the *type and purpose* of the description changes performed by the algorithm. The choice of a type determines the space of alternative descriptions explored by the algorithm, whereas the purpose specification helps to develop techniques for search in this space. We also need to analyze interactions of the new algorithm with other changer and solver algorithms. Finally, we have to identify the parts of the domain description that compose the algorithm's input.

These decisions form a high-level specification of a changer algorithm. We use specifications to summarize the main properties of description changers, thus separating them from implementation techniques. These summaries of main decisions have proved a useful development tool. In Part II, we will give specifications for all seven description changers.

When making the high-level decisions, we must ensure that they define a useful class of description changes and lead to an efficient algorithm. We compose a specification from the following five parts.

Type of description change. When designing a new algorithm, we first have to decide on the type of description change. For instance, the *Abstractor* algorithm improves the domain description by generating an abstraction hierarchy, whereas *Margie* is based on selecting primary effects of operators. In Figure 1.13, we summarize the types of description changes used in SHAPER. Note that this list is only a small sample from the space of description improvements. We will discuss some other improvements in Section 5.3.2.

Purpose of description change. Every changer algorithm in SHAPER serves a specific purpose, such as reducing the branching factor of search, constructing an abstraction hierarchy with certain properties, or improving

the effectiveness of other description changers. We express this purpose by a heuristic function for evaluating the quality of a new description. For example, we may evaluate the performance of *Margie* by the number of levels in the resulting hierarchy: the more levels, the better. We also specify certain constraints that describe necessary properties of newly generated descriptions. For example, *Margie* must preserve the completeness of problem solving, which limits its freedom in selecting primary effects.

To summarize, we view the purpose of a description improvement as maximizing a specific evaluation function, while satisfying certain constraints. This specification of the purpose shows exactly in which way we improve description and helps to evaluate the results of applying the changer. The effectiveness of our approach depends on the choice of appropriate evaluation functions, which must correlate with the resulting efficiency improvements.

Use of other algorithms. A description-changing algorithm may use subroutine calls to some problem solvers or other description changers. For example, *Margie* calls the *Abstractor* algorithm to construct hierarchies for the chosen primary effects. It repeatedly invokes *Abstractor* for alternative selections of primary effects until finding a satisfactory selection.

Required input. We identify the elements of domain description that must be a part of the changer's input. For example, *Margie* has to access the description of all operators in the domain.

Optional input. Finally, we specify the additional information about the domain that may be used in changing description. If this information is available, the changer algorithm utilizes it to generate a better description; otherwise, the algorithm uses some default assumptions. The optional input may include restrictions on the allowed problem instances, useful knowledge about domain properties, and advice from the user.

For example, we may specify constraints on the allowed problems as an input to *Margie*. Then, *Margie* passes these constraints to *Abstractor*, which utilizes them in building hierarchies (see Section 5.2). As another example, the user may pre-select some primary effects of operators. Then, *Margie* preserves this pre-selection and chooses additional primary effects (see Section 5.1).

We summarize the specification of the *Margie* algorithm in Figure 1.14; however, this specification does not account for advanced features of the PRODIGY domain language. In Section 5.1, we will extend it and describe the implementation of *Margie* in the PRODIGY architecture.

1.4.3 Search in the space of representations

When $\mathcal{S}$HAPER inputs a new problem, the control module has to determine a strategy for solving it, which involves several decisions (Figure 1.15):

1. Can $\mathcal{S}$HAPER solve the problem with the initial domain description? Alternatively, can the system reuse one of the old descriptions?

Type of description change: Selecting primary effects of operators.

Purpose of description change: Maximizing the number of levels in *Abstractor*'s hierarchy, while ensuring completeness of problem solving.

Use of other algorithms: The *Abstractor* algorithm, which constructs hierarchies for the selected primary effects.

Required input: Description of all operators in the domain.

Optional input: Restrictions on goal literals; pre-selected primary and side effects.

Fig. 1.14. Simplified specification of the *Margie* algorithm. We use specifications to summarize the main properties of description changers, thus abstracting them from the details of implementation.

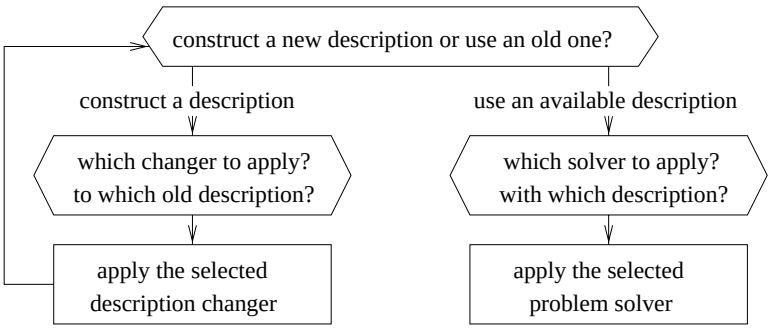


Fig. 1.15. Top-level decisions in SHAPER. The control module applies changer algorithms to improve the description and then chooses an appropriate solver.

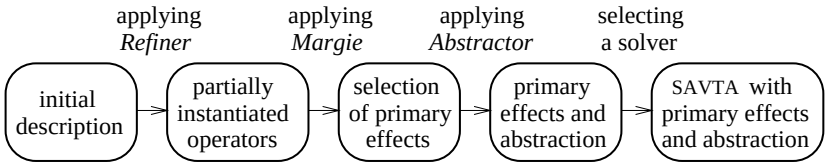


Fig. 1.16. Representation changes in the Tower of Hanoi (see Section 1.2). The system applies three description changers and identifies the most effective solver.

2. If not, what are the necessary improvements to the initial description? Which changer algorithms can make these improvements?
3. Which of the available search algorithms are efficient for the problem?

These decisions guide the construction of a new representation, which consists of an improved description and matching solver. For example, if we encode the Tower-of-Hanoi Domain using the general predicate (on <disk> <peg>) and allow the two-disk moves, the control procedure will apply three description changers, *Refiner*, *Margie*, and *Abstractor*, and then select an effective problem solver (Figure 1.16).

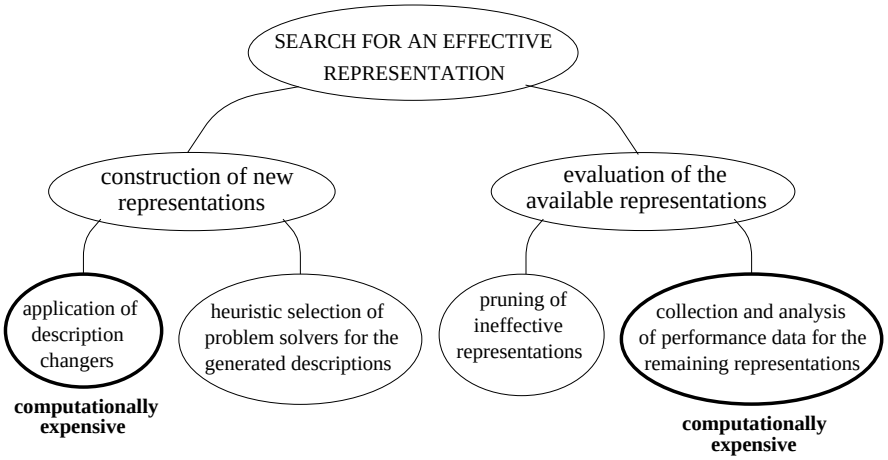


Fig. 1.17. Main operations for exploring the space of representations. SHAPER interleaves generation of new representations with testing of their performance.

When SHAPER searches for an effective representation, it performs two main tasks: generating new representations and evaluating their utility (Figure 1.17). The first task involves improving the available descriptions and pairing them with solvers. The system applies changer algorithms, using them as basic steps for expanding a space of descriptions (Figure 1.18a). After generating a new description, it chooses solvers for this description. If the system identifies several matching solvers, it pairs each of them with the new description, thus constructing several representations (Figure 1.18b).

The system evaluates the available representations in two steps (Figure 1.17). First, it applies heuristics for estimating their relative utility and eliminates ineffective representations. Second, the system collects experimental data on the performance of the remaining representations, and applies statistical analysis to select the most effective domain description and solver.

1.5 Extended abstract

The main results of the work on SHAPER include development of several description changers and a general-purpose control module. The presentation of these results is organized in four parts, as shown in Figure 1.19. Part I includes the motivation and description of the PRODIGY search. In Part II, we present algorithms for using primary effects and abstraction. In Part III, we describe the control mechanism, which explores a space of alternative representations. In Part IV, we give the results of testing SHAPER. We now give an overview of the results and summarize the material of every chapter.

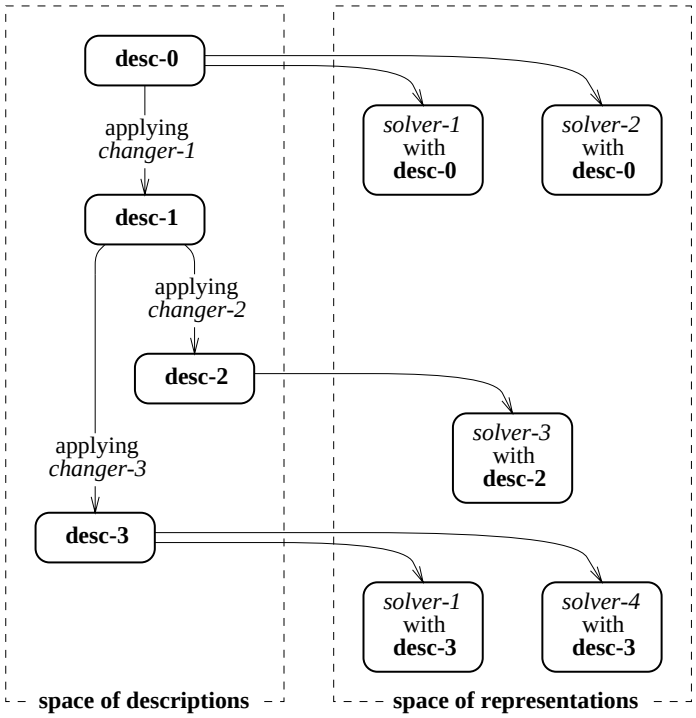


Fig. 1.18. Expanding the representation space. The control module uses changer algorithms to generate new descriptions and combines solvers with these descriptions.

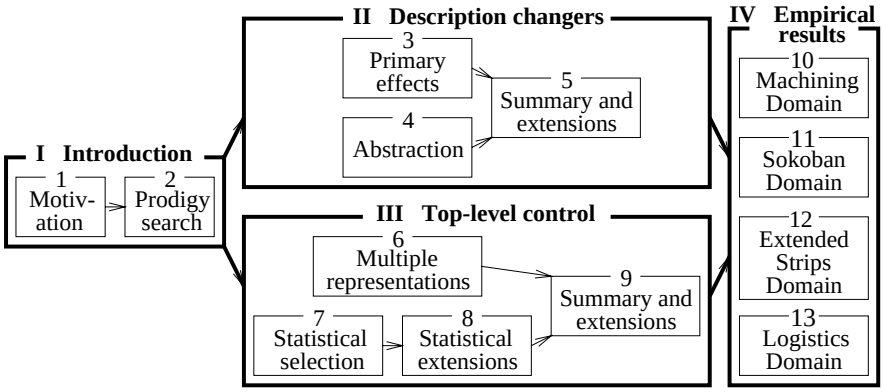


Fig. 1.19. Dependencies among different chapters. The large rectangles show the four main parts of the presentation, whereas the small rectangles are chapters.

Part I: Introduction

The purpose of the introduction is to explain the problem of representation changes and give the background results. We have emphasized the importance of appropriate representations and summarized the goals of the reported work. In Chapter 2, we describe the PRODIGY architecture, which serves as a testbed for the development and evaluation of $\mathcal{S}$HAPER.

Chapter 1: Motivation. We have explained the concept of representation and argued the need for an AI system that generates multiple representations. We have also reviewed previous work on representation changes, identified the main research problems, and outlined our approach to addressing some of them. To illustrate the role of representation, we have given an example of representation changes in the Tower-of-Hanoi puzzle.

Chapter 2: Prodigy search. The PRODIGY system is based on a combination of goal-directed reasoning with simulated execution. Researchers have implemented a series of search algorithms that utilize this technique; however, they have provided few formal results on the common principles underlying the developed algorithms. We formalize the PRODIGY search and show how different strategies for controlling search complexity give rise to different versions of the system. In particular, we demonstrate that PRODIGY is not complete and discuss advantages and drawbacks of its incompleteness. We then develop a complete algorithm, which is almost as fast as PRODIGY and solves a wider range of problems.

Part II: Description changers

We investigate two techniques for reducing the complexity of goal-directed search: identifying primary effects of operators and generating abstraction hierarchies. These techniques enable us to develop a collection of efficiency-improving algorithms, which compose $\mathcal{S}$HAPER's library of changers.

Chapter 3: Primary effects. The use of primary effects of operators allows us to improve search efficiency and solution quality. We formalize this technique and evaluate its effectiveness. First, we present a criterion for choosing primary effects, which guarantees efficiency and completeness, and describe algorithms for automatic selection of primary effects. Second, we experimentally show their effectiveness in two backward-chaining systems, PRODIGY and ABTWEAK.

Chapter 4: Abstraction. We describe abstraction for PRODIGY, present algorithms for generation of abstraction hierarchies, and give empirical confirmation of their effectiveness. First, we review Knoblock's ALPINE system, which constructs hierarchies for a limited domain language, and extend it for the advanced language of PRODIGY. Second, we give an algorithm that improves effectiveness of the abstraction generator by partially instantiating predicates in the domain encoding.

Chapter 5: Summary and extensions. We present two techniques for enhancing the utility of primary effects and abstraction. First, we describe a synergy of the abstraction generator with a procedure for selecting primary effects. Second, we give an algorithm for adjusting the domain description to a specific problem. In conclusion, we review the results of the work on description changers, summarize the interactions among the implemented algorithms, and outline some directions for future research.

Part III: Top-level control

We develop a system for the automatic generation and use of multiple representations. When the system faces a new problem, it first improves the problem description and then selects an appropriate solver. The system's central part is a control module, which chooses appropriate solvers and changers, reuses descriptions, and accumulates performance data. We describe the structure of the control module and techniques for automatic selection among the available solvers, changers, and domain descriptions.

Chapter 6: Multiple representations. We lay a theoretical groundwork for a synergy of multiple solvers and changers. First, we discuss the task of improving domain descriptions and selecting appropriate solvers, formalize it as search in a space of representations, and define the main elements of this space. Second, we develop a utility model for evaluating representations.

Chapter 7: Statistical selection. We consider the task of choosing among the available representations and formalize the statistical problem involved in evaluating their performance. We then present a learning algorithm that gathers performance data, evaluates representations, and chooses a representation for each given problem. It also selects a time limit for search with the chosen representation and interrupts the solver upon reaching this limit.

Chapter 8: Statistical extensions. We extend the statistical procedure to account for properties of specific problems. The extended learning algorithm uses problem-specific utility functions, adjusts the data to the estimated problem sizes, and utilizes information about similarity among problems.

Chapter 9: Summary and extensions. We present some extensions to the control module, which include heuristics for identifying an effective representation, rules for selecting among changers, and tools for optional user participation in the top-level control. We then summarize the main results, discuss limitations of $\mathcal{S}$HAPER, and point out related directions for future work.

Part IV: Empirical results

We give the results of applying $\mathcal{S}$HAPER to four domains: a model of a machine shop (Chapter 10), Sokoban puzzle (Chapter 11), Extended STRIPS

world (Chapter 12), and PRODIGY Logistics Domain (Chapter 13). The experiments have confirmed that the system's control module almost always selects the right solvers and changers, and that its performance is not sensitive to features of specific domains.