Task Modeling with Reusable Problem-Solving Methods

Henrik Eriksson
Yuval Shahar
Samson W. Tu
Angel R. Puerta
Mark A. Musen

Medical Computer Science Group
Knowledge Systems Laboratory
Stanford University School of Medicine
Stanford, California 94305-5479, U.S.A.

June 1, 1994

Abstract

Problem-solving methods for knowledge-based systems establish the behavior of such systems by defining the roles in which domain knowledge is used and the ordering of inferences. Developers can compose problem-solving methods that accomplish complex application tasks from primitive, reusable methods. The key steps in this development approach are task analysis, method selection (from a library), and method configuration. PROTÉGÉ-II is a knowledge-engineering environment that allows developers to select and configure problem-solving methods. In addition, PROTÉGÉ-II generates domain-specific knowledge-acquisition tools that domain specialists can use to create knowledge bases on which the methods may operate.

The *board-game method* is a problem-solving method that defines control knowledge for a class of tasks that developers can model in a highly specific way. The method adopts a conceptual model of problem solving in which the solution space is construed as a "game board" on which the problem solver moves "playing pieces" according to prespecified rules. This familiar conceptual model simplifies the developer's cognitive demands when configuring the board-game method to support new application tasks. We compare configuration of the board-game method to that of a chronological-backtracking problem-solving method for the same application tasks (for example, Towers of Hanoi and the Sisyphus room-assignment problem). We also examine how method designers can specialize problem-solving methods by making ontological commitments to certain classes of tasks. We exemplify this technique by specializing the chronological-backtracking method to the board-game method.

1 Reusable Components for Knowledge Engineering

During the past decade, developers of knowledge-based systems have realized that the representations that they use to encode expertise at the level of rules or frames do not provide sufficient abstraction for the design of large, complex systems. During this period, researchers have begun to consider frameworks that can capture the behaviors of such systems more abstractly than can rules and frames. Chandrasekaran [2; 3], for example, has proposed the use of domain-independent problem-solving methods, or *generic tasks,* for typical problems as a basis for the design of knowledge-based systems. Clancey [5] has analyzed retrospectively several rule-based systems, and has identified an inference structure for *heuristic classification.* McDermott and his colleagues have developed a series of special-purpose problem solvers with corresponding knowledge-acquisition tools [29]. In their approach, the problem solvers use different *role-limiting problem-solving methods* as the reasoning strategy. For instance, VT [26] uses a *propose-and-revise* method for accomplishing the task of elevator configuration, and MOLE [14] uses a *cover-and-differentiate* method for classification. Such methods make use of limited knowledge roles in the sense that they identify explicitly the different ways in which the problem solver uses inferences from the knowledge base. If the developer can use a preexisting role-limiting method, the role-limiting approach reduces the development task to one of identifying what domain knowledge is required to fill each role.

The role-limiting approach, however, assumes that the problem-solving behavior of a knowledge-based system can be defined in domain-independent terms [29]. For role-limiting methods to be reusable across application domains, they must be general, which often means that it is difficult to match a method with a particular application task, because there can be a significant semantic gap between a general method and an application task. Problem-solving methods designed by researchers and developers cannot easily be used for other purposes, or be reused in other similar projects [33]. Current research in knowledge sharing and reuse proceeds along two avenues: reusable *ontologies* (which define concepts and their relationships) and reusable problem-solving *methods* (which defines operations for problem solving) [37]. Currently, our research is primarily concerned with reusable components for the operational aspects of knowledge-based systems—that is, reusable problem-solving *methods.*

Identifying the task of the knowledge-based system is an important first step toward finding an appropriate problem-solving method. In this context, a *task* is the real-world activity that the knowledge-based system should accomplish. Developers must identify, at least partially, the task of the system they are designing before they can select and custom tailor preexisting methods. This *task analysis* leads to a system-role description in terms of the domain for the system, which serves as the basis for the *selection* of problem-solving methods that accomplish the task [21; 28], and for the *configuration* of the methods selected for the task instance. Many researchers have pointed out similarities among application domains, and among the methods that can be used for problem solving in these domains [5; 20; 23; 29; 32; 44]. Increasingly, these researchers have noted that such similarities can be used as a foundation for developing reusable methods and other reusable components for knowledge-based systems [3; 4; 28; 32; 39; 44]. One of the most important lessons learned from the work discussed is the importance of the developers' conceptual models of problem solving for method reuse. In Sections 1.1 through 1.3, we shall provide a brief historic background to our work, and shall introduce the work on reusable problem-solving methods presented in this article.

1.1 Development Environments: Background

In the mid-1980s, our laboratory developed OPAL [34], a domain-specific knowledge-acquisition tool that allows physicians to enter cancer-treatment plans for the ONCOCIN therapy advisor [47]. The principal advantage of tools such as OPAL is that they are custom tailored precisely for the application task and for the problem-solving method used to accomplish that task. The major weakness of domain-specific tools, however, is that they are useful for developing systems in only one domain. To remove this limitation, we developed the *metatool* PROTÉGÉ [30; 31], which allows developers to generate knowledge-acquisition tools similar to OPAL for other application domains. PROTÉGÉ supports the generation of knowledge-acquisition tools for the role-limiting problem-solving method used by ONCOCIN—*episodic skeletal-plan refinement* [47]. Although PROTÉGÉ demonstrated the feasibility of automated generation of knowledge-acquisition tools from instantiation of the data requirements of a method, PROTÉGÉ cannot support the development of knowledge-based systems that require problem-solving methods other than skeletal-plan refinement [35]. Recognizing the limitations of a single problem-solving method, we are designing PROTÉGÉ-II, a system that supports more general task-oriented knowledge engineering [38; 39].

1.2 Tasks and Problem-Solving Methods in PROTÉGÉ-II

The PROTÉGÉ-II system provides a knowledge-engineering environment in which the developer can specify tasks, and can select problem-solving methods from a library of reusable methods. The PROTÉGÉ-II approach distinguishes between tasks and problem-solving methods [32; 38]. Tasks are real-world functions that the knowledge-based system is supposed to discharge. Examples of such tasks follow: (1) given a set of symptoms for a faulty device (e.g., manual observations and instrument readings), produce a diagnosis and a remedy; (2) given an initial state and a goal, produce a plan (i.e., a series of operations) that will accomplish the goal; and (3) maintain at steady values certain measurements and indicators of a manufacturing process over time (i.e., control). In the PROTÉGÉ-II approach, the developer analyzes the application task manually, and uses the PROTÉGÉ-II system to identify appropriate methods in the library and to configure the methods to perform the task.

Problem-solving methods can be seen as abstract models of how to solve certain problems [29]. In PROTÉGÉ-II, *methods* are actions that accomplish tasks. Examples of such problem-solving methods are (1) state-space search by chronological backtracking, (2) classification (e.g., classification of faults given symptoms), (3) reactive planning [7; 16], (4) skeletal-plan refinement [47], (5) temporal abstraction [42; 43], and (6) propose-and-revise methods for configuration [26]. Often, a specified task can be accomplished by several methods. For instance, we can perform troubleshooting tasks by matching and classifying faults, or by using model-based reasoning. The selection of a method may depend on factors beyond the task specification, such as availability of expertise, time and space requirements for computations, and compatibility with other cooperating methods.

Methods can delegate problems as *subtasks* to be solved by other methods. We use the term *mechanism* for primitive methods that cannot be decomposed into subtasks (Figure 1). Methods solve the problem imposed by their task, and methods may pose new subtasks in the process of accomplishing the overall task. Because mechanisms are capable of accomplishing the task without delegation, the developer can regard the mechanisms as black boxes that cannot be decomposed further.

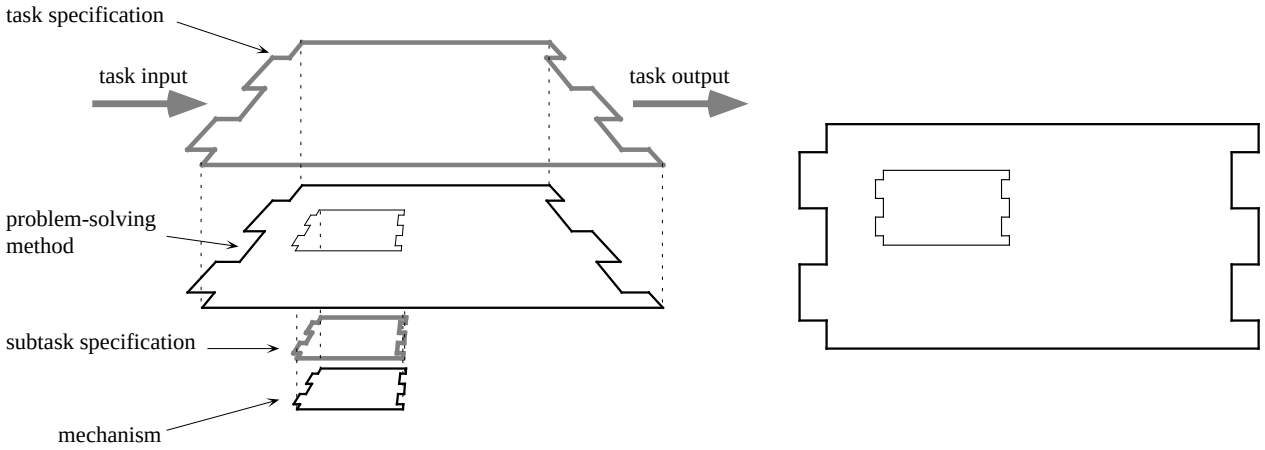

Figure 1: Tasks and methods. (a) The relationships among task specifications, methods, subtask specifications, and mechanisms in PROTÉGÉ-II. (b) The configured problem solver during execution of the target knowledge-based system.

The performance of knowledge-based systems is critically dependent on the domain expertise available to the problem solver. For example, methods for heuristic classification rely on domain-specific classification knowledge. Knowledge acquisition from domain experts is an important technique for the development of such knowledge bases. Knowledge-acquisition tools based on strong domain models provide environments in which experts can enter knowledge according to a conceptual model of the domain [8; 10; 11; 12; 34]. In addition to supporting the development of problem solvers for knowledge-based systems, PROTÉGÉ-II generates domain-specific knowledge-acquisition tools that elicit the expertise required by the problem-solving methods to perform the latter's tasks. Figure 2 shows the overall architecture of the PROTÉGÉ-II environment.

1.3 A Study in Method Reuse

The ultimate goal of our work is to develop techniques for real-world method reuse. Because it is difficult to explore various approaches to method reuse for full-scale systems, we study principles for reuse on well-defined, standard problems. In particular, we focus on the description and representation of problem-solving methods, and on the process of method reuse. In the following discussion, we shall use two exemple problems to illustrate task modeling, method selection, and method configuration. In addition to analyzing modeling of these problems, we examine how well methods map onto the domain tasks, and how reusable methods support the developer. In particular, we compare several problem-solving methods for the towers-of-Hanoi task, such as state-space search by chronological backtracking and the classic recursive solution (recursive-task decomposition). We present a *board-game* method that can solve a class of problems in which playing pieces move between board locations under certain constraints. This method embraces board games as a conceptual model for developers to understand the problem-solving strategy employed by the method, and as a model for method configuration. Although we use these methods to model small-scale tasks, we believe that

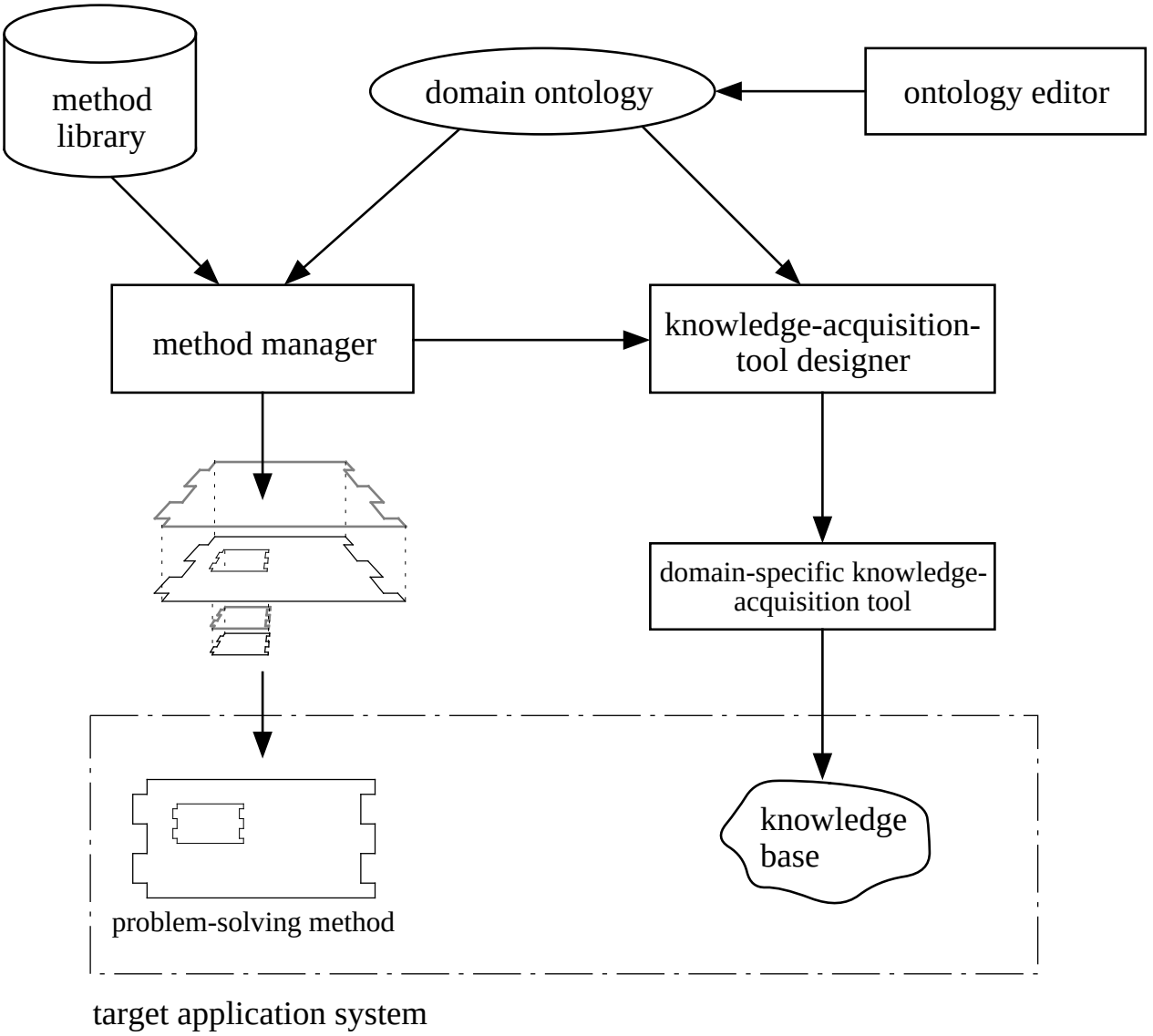

Figure 2: The architecture of the PROTÉGÉ-II development environment. Development tools are shown as rectangles. The developer uses the *method manager* to retrieve methods from the *method library,* and to configure the methods for their tasks [38]. Moreover, the developer uses the *knowledge-acquisition-tool designer* to generate a *domain-specific knowledge-acquisition tool,* which elicits the appropriate domain knowledge, and generates *knowledge bases* for the *problem-solving methods* [13]. The developer uses an *ontology editor* to create domain ontologies, which are used by the method manager and the knowledge-acquisition-tool designer. The target application system consists of configured problem-solving methods, domain ontologies, and appropriate knowledge bases.

many of the principles for reuse applied here can be used ultimately for realistic problem domains.

The remainder of this article is organized as follows. Section 2 presents problem definitions and task analyses of two example problems: the towers-of-Hanoi and Sisyphus room-assignment problems [25]. In Section 3, we discuss the selection of a problem-solving method for a task. Section 4 examines configuration of methods for new tasks. In Section 5, we discuss the relationship between ontologies and problem-solving methods. Section 6 discusses the knowledge acquisition required for these method configurations. In Section 7, we present the results from this examination of reuse of problem-solving methods; in Section 8 we discuss related work. Section 9 summarizes and presents the conclusions of the article.

2 Task Analysis and Example Tasks

Each time that developers are confronted with a new task, they must understand and model the task before they can select and configure a reusable problem-solving method. Task analysis is a *modeling* activity; the designer identifies the problem, as well as the inputs and outputs of the problem-solving process. The result of the task analysis can be a formal problem specification; more often, it is an initial informal description of the problem that can be used as a basis for identification of an appropriate problem-solving method.

Once the developers have analyzed the task to a point where the input–output relationship and the knowledge available are identified, they can form a hypothesis about the appropriate methods. Task analysis, however, does not stop where method selection begins. Task analysis is a continuous activity in the sense that the developers must be prepared to revise and extend their model of the task as they gain more insight into the problem. For example, method selection might reveal that the developers have incomplete knowledge of the task, and that they must continue analyzing the task, because their candidate problem-solving method needs additional knowledge.

The details of the task-analysis process are beyond the scope of this discussion, but there are many approaches to task analysis described in the literature. One of the most prominent approaches is KADS [49], which provides a layered framework for models of expertise. The result of the classic KADS methodology is not an executable system, but rather is a conceptual model of expertise. There is significant ongoing work that involves the formalization of KADS models, and the implementation of executable KADS models [15; 19].

We shall use the towers-of-Hanoi problem and the Sisyphus room-assignment problem as the main illustrative examples. The towers-of-Hanoi task is an artifact, whereas the room-assignment task represents a problem class somewhere in between artificial and realistic tasks. We shall provide the result of our task analysis for the towers-of-Hanoi and room-assignment problems as a background to the following sections on method selection and configuration.

2.1 The Towers of Hanoi Task

The towers-of-Hanoi task is interesting as a case study of a tradeoff of space and time resources with more task-specific knowledge. It demonstrates several possible task-level indices that can be used to select candidate problem-solving methods from a library. These indices characterize different dimensions of the problem and of its potential solutions.

(a) (b)

Figure 3: The classical three-disk towers-of-Hanoi problem. (a) The initial state. (b) The goal state.

The towers-of-Hanoi problem is a game in which pieces move between locations. In the towers-of-Hanoi game, there are k locations, called *pegs*. There are n pieces, called *disks*. The disks reside at the pegs, and can move between pegs. A move operation consists of popping one disk from one location's stack and pushing it onto another location's stack. Disks have various sizes, and a local configuration constraint stipulates that disks in every location must form a *tower* ordered by size, the largest disk being on the bottom, the smallest on top. The initial and goal configurations can be arbitrary. In what we shall call the *classic version* of the towers-of-Hanoi problem (shown in Figure 3), there are three pegs, and the initial and goal configurations are tower configurations.

Two important parameters that developers must resolve during the task analysis are the premise and the result of the task (i.e., the run-time input and output of a problem solver that accomplishes the task). The potential inputs to a problem solver for the classic version are the number of disks n. Alternative formulations of the game allow arbitrary initial and goal states by assuming that these states are input to the problem solver, or allow any number k of pegs. Moreover, if we generalize the game definition further, certain constraints on the game can be input to the problem solver. In the classic version, and in most alternative formulations of the game, the result sought is a *plan*—that is, a sequence of moves that takes the board from its initial state to its goal state. Also, if the goal of the game is expressed as a predicate, rather than as an explicit state, the final state can be part of the result. The specific input-and-output requirements of the task depend, of course, on the context in which the problem solver will be used, such as the user community. Nevertheless, these input-and-output requirements determine what assumptions the problem-solving methods can make, and, therefore, what methods we can use to accomplish the task.

2.2 The Sisyphus Room-Assignment Task

The Sisyphus[1] room-assignment task is a standard task that is used by researchers in knowledge acquisition and reusable problem-solving methods to compare their modeling approaches [25]. Thus, the primary purpose of the Sisyphus problem formulation is to compare different approaches; it is *not* to find the best solution to the problem per se. The methods that researchers have used to model the Sisyphus room-assignment task range from repeated application of heuristic classification to simulated annealing [25].

[1]The Sisyphus experiment to compare different approaches to knowledge acquisition is named after the legendary king of Corinth, who was condemned to roll continuously a heavy rock up a hill in Hades, only to have the rock roll down again.

Table 1: Excerpt from the Sisyphus problem-solving transcript [25]. *Source:* The Sisyphus problem statement by Marc Linster, Digital Equipment Corporation; used with permission.

Step	Words of the expert	Annotations
1.	Put Thomas D. into room C5-117	The head of group needs a central room, so that he is as close to all the members of the group as possible. It should be a large room. This assignment is defined first, as the location of the room of the head of group restricts the possibilities of the subsequent assignments.
2.	Put Monica X. and Ulrike U. into room C5-119	The secretaries' room should be located close to the room of the head of the group. Both secretaries should work together in one large room. This assignment is executed as soon as possible, as its possible choices are extremely constrained.
3.	Put Eva I. into C5-116	The manager must have maximum access to the head of the group and to the secretariat. At the same time, she should have a centrally located room. A small room will do. This point is the earliest one at which this decision can be made.

In essence, the task is to assign persons to office rooms under certain constraints. A research group is moved to a new location, and rooms must be assigned to persons. The role of the knowledge-based system is to allot rooms to persons in the group given information about the staff, descriptions of the rooms, and a set of constraints. Individuals have their own professional characteristics and personal preferences (e.g., professional role, current project, or use of tobacco). The information available at run time consists of a set of person descriptions and a set of room descriptions.[2] In the Sisyphus problem description [25], a document widely circulated in the knowledge-acquisition community, knowledge is expressed as a transcript of a problem-solving session, during which a human expert assigns persons to rooms. Each assignment step is explained by a brief comment in natural language (see Table 1 for an excerpt of the transcript). Figure 4 shows a sample floor plan. Tables 2 and 3 show attributes for persons and rooms derived from the Sisyphus problem definition.

Even for a relatively simple task, such as the room-assignment problem, many design decisions must be made in the task-analysis phase. The role of a knowledge-based system in this domain is to *replace* the current expert in room assignment. (Alternative system roles could be to simulate consequences of various assignments, to train novices in room assignment, or to critique solutions.) The descriptions of the staff and of the rooms are input to the system; the output of the system is a mapping from persons to rooms (i.e., a set of person-to-room assignments). Another important factor for the task analysis is the availability of expertise. Domain knowledge is available in the form of a transcript from a problem-solving session.

[2] These data are clearly variable; thus, they should be used as task input if the system will be used for assigning persons to rooms for several groups. Indeed, most approaches to the Sisyphus problem regard person and room descriptions as run-time input [25].

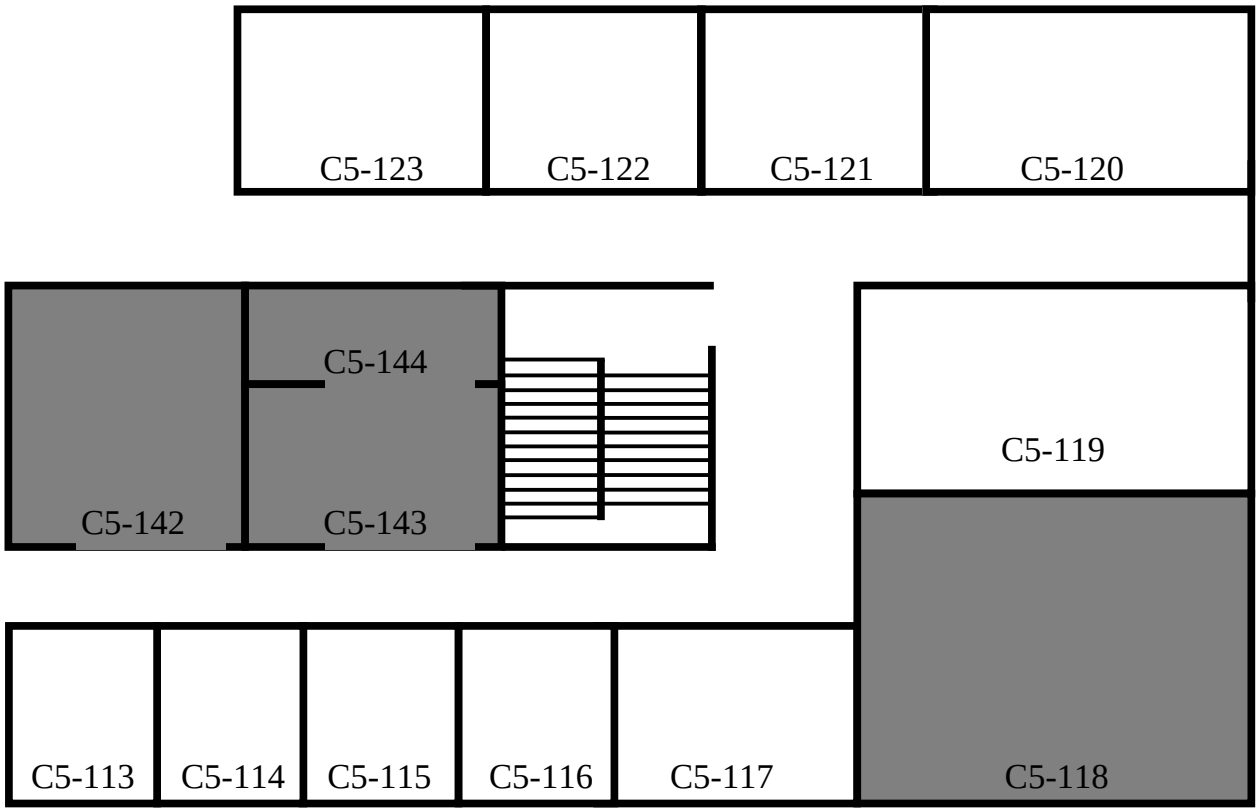

Figure 4: A sample floor plan in the Sisyphus room-assignment problem. The task is to assign persons to rooms under certain constraints. *Source:* The floor plan was provided by Marc Linster, Digital Equipment Corporation; used with permission.

Table 2: Attributes of persons in the Sisyphus room-assignment problem.

Attribute	Description
`name`	what the name of the person is
`professional role`	what the professional role of the person (e.g., researcher or secretary) is
`smoker`	whether the person smokes tobacco
`hacker`	whether the person is doing programming
`works with`	who the person's coworkers are
`single room`	whether the person is eligible for a single room

Table 3: Attributes of rooms in the Sisyphus room-assignment problem.

Attribute	Description
`name`	what the name or number of the room is
`size`	what the size of the room (small or large) is
`central`	to what degree the room is located centrally in the suite

3 Method Selection

In this section, we shall discuss several methods that developers can use to solve the towers-of-Hanoi and Sisyphus tasks. Our purpose is to provide a demonstration of the tradeoffs associated with method selection, and of the effect of increased insight into the task on method selection and configuration.

3.1 Selection Considerations

Although it is difficult to make a comprehensive list of factors to consider in method selection, we can identify a set of recurring factors that are applicable to most tasks. Common factors to consider in the selection of a problem-solving method include:

1. *Input and output of the task.* What information is available at run time? What is the run-time output? Are explanations required?

2. *Knowledge available.* What type of knowledge can be acquired from domain specialists?

3. *Solution quality.* Is an optimal solution required, or is an approximation sufficient?

4. *Computational and space complexity.* What are the resources available in terms of time and space?

5. *Method flexibility.* Is the task likely to be modified during development and maintenance? What flexibility in terms of reconfiguration of the method for modified tasks is required?

Even for relatively simple tasks, such as the towers-of-Hanoi problem, there are many important factors that determine the method selection. For example, the initial and goal states may be available at run time only, or these states may be given as part of the task definition. In the latter case, the developer can select a method that takes advantage of the state information in the problem-solving strategy.

Ideally, the method selected consists of executable code that can be configured for the task. If the developer cannot find an appropriate method or can construct one from a set of primitive methods, literature references can serve as a helpful inspiration for the design of new methods (which may be included eventually in the method library).

3.2 Method Selection for the Towers-of-Hanoi Task

In the classic version of the towers-of-Hanoi problem, the initial and goal states are given as part of the problem definition. Also, the problem definition does not provide any move strategies (i.e., domain knowledge); it is up to the player to plan the moves. The solution quality, computational and space complexity, and method flexibility are not provided in the classic version per se. These factors depend on the context in which the problem solver should operate. We shall describe briefly a few methods that can be used to accomplish the towers-of-Hanoi task.

1. *Chronological-backtracking method.* The chronological-backtracking method searches the space of states for a permissible sequence of states that will reach the goal state from

the initial state. The method backtracks as necessary during the search process. Method configuration involves the representation of states, the specification of initial and goal states, and the definition of transitions from states to subsequent states. Appendix A.1 discusses the details of this method.

2. *Recursive task decomposition.* Recursive task decomposition is an approach that breaks down recursively the overall tasks into smaller subtasks. The decomposition continues until the subtasks can be accomplished by a primitive method. Mapped to the towers-of-Hanoi task, this approach represents the classic recursive solution of the problem. Appendix A.2 discusses recursive task decomposition in terms of the towers-of-Hanoi problem.

3. *Iterative and piece-oriented methods.* The iterative and piece-oriented methods require the developer to define a set of precise rules for generating an appropriate sequence of state transitions. These methods rely extensively on domain knowledge, rather than on problem decomposition. However, they also enjoy certain unique advantages over the more general methods. Appendices A.3 and A.4 discuss move rules for the towers-of-Hanoi task in the context of these methods.

4. *General-task decomposition.* General-task decomposition is similar to the method of recursive task decomposition in that it decomposes the task into subtasks. General-task decomposition, however, takes advantage of a subtask that transfers any state to a single-tower state (i.e., a canonical state). This approach is similar to *macro operators* [22]. Appendix A.5 describes this method in detail.

Chronological backtracking is a general method that can provide solutions for several versions of the task, including nonclassical towers-of-Hanoi games (such as alternative initial and goal configurations). These solutions are nonoptimal, however, in terms of computational complexity and the number of required moves. By making further commitments to the task and taking advantage of additional domain knowledge, we can reduce the upper limit for the computational complexity. Thus, the more specific solutions can be viewed as task-specific heuristics for the general chronological-backtracking method. By using perfect knowledge, we can completely avoid backtracking, and can guarantee an optimal solution. In general, however, we might have more than three pegs, and we might start or end with any state; the domain definition of a legal move might be different, too (e.g., it might be legal to move whole parts of a tower during one move). Although the task-specific methods are more usable than is chronological backtracking with respect to alternative problem variants, they are not reusable across different tasks.

3.3 Method Selection for the Sisyphus Room-Assignment Task

As discussed in Section 2.2, the input to the room-assignment problem solver consists of the rooms and the persons to assign, and the output is a legal assignment of persons to rooms. Furthermore, the domain knowledge is provided as a problem-solving transcript (see Table 1). In addition to the factors listed in Section 3.1, the developer must consider the problem and knowledge representation in the method selection. The developer must be able to map the representations used by potential methods to representations appropriate for the task.

Another question is whether a computer-based problem solver should follow the expert's reasoning precisely. A method that follows the transcript in Table 1 exactly will make all decisions in the same order as the expert does. Thus, the method will be a strict model of the problem-solving strategy used by the expert (or, more precisely, the strategy indicated by the utterances captured in the protocol). An alternative approach is to decouple the method from the transcript completely. The Sisyphus task potentially can be modeled according to both approaches. The selection of an appropriate method depends mainly on the domain knowledge available.

We use the *board-game* method to model the Sisyphus room-assignment task. The board-game method can accomplish tasks that the developer can model as a set of *pieces* that move among *locations* under certain constraints. Because we can view the room-assignment problem as a game where persons move among rooms, we can cast the method to the room-assignment task. (Initially, persons are located outside the building. Each person then moves into a room under the assignment constraints.) Therefore, our motivation for using the board-game approach is that this method provides a conceptual model of problem solving that we can map readily to the task. Also, the board-game method can model the towers-of-Hanoi task by defining the move rules for how disks can move among pegs. Section 3.4 describes the board-game method in detail.

3.4 Board-Game Method

The concept of the board-game method is to view a problem as a board game in which *pieces* move between *locations* (Figure 5). We assume that the game has a fixed number of pieces and locations. More than one piece can be moved to the same location simultaneously, and, if required, the pieces at each location can be ordered. The notion of *states,* which is an important part of the configuration of the chronological-backtracking method, exists in the board-game method, but the notion of *moves* dominates the configuration of the board-game method. In the general case, several pieces may be transferred in each move—that is, a move may consist of a number of *actions.* Legal moves are defined by constraints on how pieces move between locations. Here, there is no explicit notion of a transition function that transforms a given state into the next state. Rather, actions represent the withdrawal of a piece from the source location and the deposit of that piece at the target location. This commitment restricts the types of tasks that the board-game method can perform, but makes task modeling easier for the class of tasks supported by the method.

Moves can be performed only when certain conditions are met—for example, the game rules might stipulate that the target location for the move must be empty. In addition to the move conditions, there are constraints expressing the legal situations in the game. For instance, a move that is legal superficially may lead to a forbidden situation in the game. We refer to such situations as constraint violations or *contradictions* in the game.

Before we can define the board-game method, we must establish how states represent board configurations. Let R be the set of *potential* states of the board game. A potential state is any (not necessarily legal) assignment of pieces to locations. A state $S \in R$ is characterized by

$$\begin{cases} \text{locations } V = \{v_i \,|\, i = 1, \ldots, k\}, \\ \text{pieces } P = \{p_j \,|\, j = 1, \ldots, n\}, \\ \text{potential board configuration } \mathrm{C}(S) \subseteq P \times V, \end{cases}$$

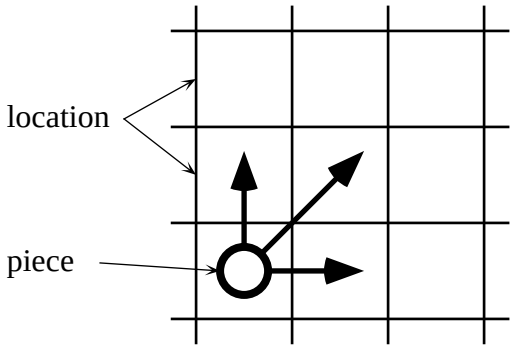

Figure 5: In the board-game method, game *pieces* move between board *locations*.

where $(p_j, v_i) \in \mathrm{C}(S)$ means that piece p_j is at location v_i in state S. Naturally, not all the potential states of a given game are legal. Let F be a set of *forbidden* states in the game. A state S satisfies the predicate $\mathsf{contradiction}(S)$ if $S \in F$. The set of legal states R_{legal} in the game is defined as $R_{\mathrm{legal}} = R \setminus F$ (where the \-operator denotes set difference). By defining states as sets of assignments, we can represent impermissible and unusual situations, such as when a piece exists in multiple locations simultaneously—a condition that might arise in certain board-game tasks. To simplify the following definitions, we shall use the notation $v_i(S)$ to denote the pieces at location v_i in state S. The function $v_i(S)$ can be defined in terms of $\mathrm{C}(S)$ as $v_i(S) = \{p \,|\, (p, v_i) \in \mathrm{C}(S)\}$. Certain board-game tasks might require that the pieces at a particular location are ordered, in which case the value of $v_i(S)$ could be defined as a list rather than as a set.

We can now define the board-game method in terms of the chronological-backtracking method. The this method requires that we define a *transition function (T-function)* that produces subsequent states from the current state (see Appendix A.1). A T-function adapted for board games can handle the generation of subsequent states in chronological backtracking. The T-function for the board-game method can be defined as

$$
\begin{aligned}
T(S) = \{S' \quad | \quad & \forall p, v_i, v_j : \; p \in P \wedge v_i, v_j \in V \wedge \\
& p \in v_i(S) \wedge \\
& \mathsf{possible\_move}(S, p, v_i, v_j) \wedge \\
& \mathsf{transfer}(S, p, v_i, v_j, S') \wedge \\
& \neg\mathsf{contradiction}(S') \,\}, \\
& \text{where } i = 1, \ldots, k; j = 1, \ldots, k; i \neq j.
\end{aligned}
$$

The function $\mathrm{T}(S)$ returns the union of the subsequent states that results from application of all possible moves to the current state S. The new state S' is the result of performance of a move from the location (state variable) v_i to the location v_j. The predicate $\mathsf{possible\_move}(S, p, v_i, v_j)$ defines when it is possible to move piece p on location v_i in state S to location v_j. The predicate defines all the possible moves of the piece p in the state S. Note that possible moves may still result in illegal states, as defined by the predicate $\mathsf{contradiction}(S)$, in which case the board-game method will not perform the move. From the definition of the function $\mathrm{T}(S)$, it follows that moves and actions can be performed by

removal of pieces from one location and deposit of the same pieces to another location. The predicate $\mathsf{transfer}(S, p, v_i, v_j, S')$ defines the new state S' in terms of the current state S, the piece p, the current location v_i, and the target location v_j. For the board-game method, we define the transfer predicate as

$$\begin{aligned}
&\forall S, p, v_i, v_j, S' : \\
&\qquad \mathsf{transfer}(S, p, v_i, v_j, S') \leftarrow \\
&\qquad\quad v_i, v_j, v_k \in V \wedge \\
&\qquad\quad v_j(S') = \{p\} \cup v_j(S) \wedge \\
&\qquad\quad v_i(S') = v_i(S) \setminus \{p\} \wedge \\
&\qquad\quad \forall v_k((v_k \neq v_i) \wedge (v_k \neq v_j) \rightarrow (v_k(S) = v_k(S'))).
\end{aligned}$$

Note that certain classical artificial-intelligence problems—such as the frame and ramification problems—do not arise here, because we make strong assumptions about the nature of a move. For example, we assume that the moves have no side effects other than moving pieces between locations, and that the consequences of a move are well defined. Also note that the strong assumptions make it difficult to model certain games where moves have side effects, and where pieces can change type during the game (e.g., when a chess pawn reaches the end of the board and is promoted into a queen).

4 Method Configuration and Specialization

Before we can use a method to accomplish a task, we must *configure* the method to handle the particular task instance. Because it is impossible to create a library of reusable methods that will fit every task precisely, we use generic methods that handle generalized tasks. Developers can then configure the method selected to solve the domain task, and method designers can *specialize* methods to match a specific class of tasks.

4.1 Configuration

Method configuration is largely a matter of (1) selecting mechanisms (or methods) for a method's subtasks (see Figure 1), and (2) defining the mapping between method terms and domain terms. The method designer defines the method such that there are appropriate subtasks where alternative mechanisms can be used. Typically, the method designer recommends a set of mechanisms for each subtask. The developer can then select a method from the set of recommended ones, or can choose a method that was not anticipated by the method designer. By selecting different methods for performing the subtasks, the developer can cause radically different behavior of the method. Therefore, it is the responsibility of the method designer to identify subtasks that enable reusability for a large class of tasks, while providing guidance for task modeling.

When the subtasks have been modeled, the developer proceeds with the definition of the mapping between method and domain concepts. The board-game method, for example, uses concepts such as *pieces* and *locations,* whereas the room-assignment task is concerned with concepts such as *persons* and *rooms.* Likewise, the developer must map the concepts of the subtasks to the concepts supported by the mechanisms performing the subtasks. Section 5 discusses in detail the relationship between problem-solving methods and domain ontologies.

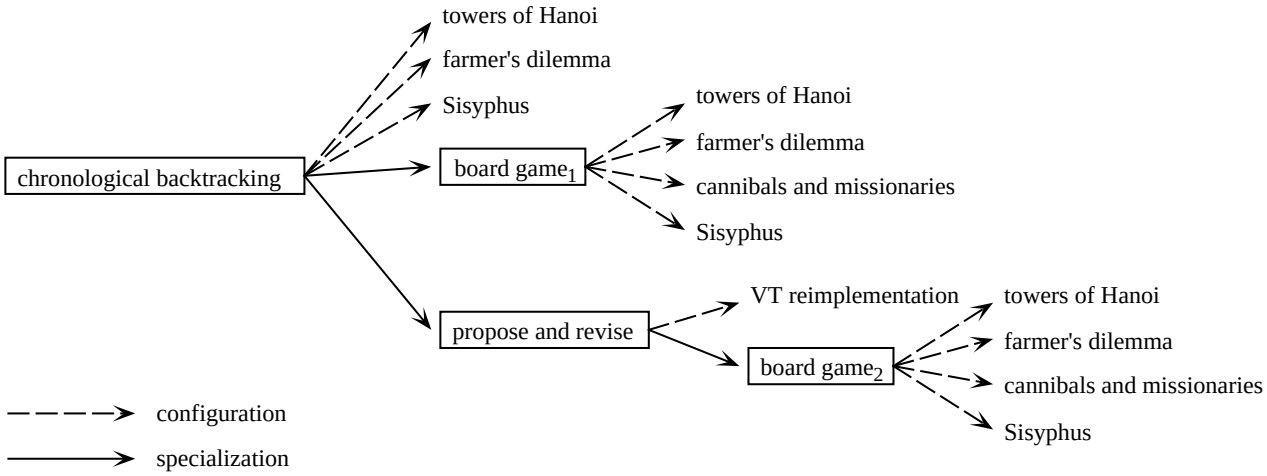

Figure 6: Specialization and configuration of methods. In addition to the towers of Hanoi and Sisyphus tasks, we have experimented with method configuration for the farmer's dilemma, cannibals-and-missionaries, and VT tasks. We have used the chronological-backtracking method to model three tasks (towers of Hanoi, farmer's dilemma, and Sisyphus), as well as to implement the board-game and propose-and-revise methods. In turn, we have used the board-game method to reimplement the three tasks, as well as the cannibals-and-missionaries problem. The methods board game_1 and board game_2 are two alternative implementations that are based on the chronological-backtracking and propose-and-revise methods, respectively.

To experiment with method configuration, we have (1) implemented in CLIPS[3] [36] the methods discussed and (2) configured these methods to perform example tasks, such as the towers-of-Hanoi and Sisyphus room-assignment tasks. In addition to chronological-backtracking and the board-game methods, we have developed and configured the *propose-and-revise* method for the VT task (i.e., elevator configuration) [27; 40]. Figure 6 illustrates the relationship between the method configurations. Note that we have modeled some of these tasks using different methods. Section 7 provides a comparison of the different modeling approaches.

We shall discuss the details of two interesting method configurations. In Section 4.1.1, we use the general chronological-backtracking method to model the towser-of-Hanoi task. This example illustrates the use of a general method for a relatively simple task. As shown in Figure 6, we can configure chronological-backtracking for the Sisyphus task, and for the implementation of the board-game and propose-and-revise methods, but these configuration and too lenghth to discuss in detail (see Section 7). In Section 4.1.2, we examine the configuration of the board-game method for the room-assignment task. This example illustrates how high-level concepts, such as *possible moves* and game *contradictions,* are used in the method configuration.

[3] CLIPS is a programming language that supports object-oriented programming and production rules. CLIPS has a Lisp-like syntax, is implemented in C, and runs on multiple platforms.

4.1.1 Configuration of the Chronological-Backtracking Method for the Towers of Hanoi Task

To configure the chronological-backtracking method for the towers-of-Hanoi task, the developer must define the disk-moving task, and must specialize chronological backtracking as the solution method for the task. The chronological-backtracking method requires that we define the concept of *state* and an *equality predicate* for states. To detect and avoid circularities in the state space, the method uses the equality predicate to check whether the current state on its search path is equal to a state that has already been encountered on the path. To model the towers-of-Hanoi problem, we represent the pegs as a set of state variables, $\{v_1, v_2, \ldots, v_n\}$, each of which can take as its value a list of positive integers or the empty list. The integers represent the disks on the peg, and the integer values represent the sizes of the disks. The empty list indicates that no disk is placed on the peg. A state S in the disk-moving task is represented by the *values* of all the state variables $\{v_1(S), v_2(S), \ldots, v_n(S)\}$. Two states are considered equal if the values of all state variables in one state are equal to the values of the corresponding state variables; therefore, the equality predicate can be defined as

$$\forall S, S', i : \mathsf{equal}(S, S') \leftarrow v_i(S) = v_i(S').$$

In the towers-of-Hanoi disk-moving task, we define the next possible states as the result of moving one disk from one peg to another. We use the domain-specific predicate $\mathsf{move\_disk}(S, v_i, v_j, S')$ to specify the move of the top disk of peg i to the top of peg j. The variable $v_i(S)$ holds a set of integers representing the disks at peg i in state S. The function $\mathsf{min}(v_i(S))$ returns the smallest integer in this set—the integer representing the smallest, topmost disk at the peg i. The predicate $\mathsf{move\_disk}(S, v_i, v_j, S')$ can be defined as

$$\begin{aligned}
&\forall S, v_i, v_j, S' : \\
&\qquad \mathsf{move_disk}(S, v_i, v_j, S') \leftarrow \\
&\qquad\quad v_i, v_j, v_k \in V \ \wedge \\
&\qquad\quad v_i(S) \neq \emptyset \ \wedge \\
&\qquad\quad v_j(S') = \{\mathsf{min}(v_i(S))\} \cup v_j(S) \wedge \\
&\qquad\quad v_i(S') = v_i(S) \setminus \{\mathsf{min}(v_i(S))\} \wedge \\
&\qquad\quad \forall v_k (v_k \neq v_i \wedge v_k \neq v_j \rightarrow v_k(S) = v_k(S')).
\end{aligned}$$

We can now define $\mathrm{T}(S)$ as the union of the results of applying all possible moves from each state variable. That is, for each state variable v_i, the set of i and j permutations, where $i \neq j$, represents all the possible actions of moving one disk from peg i to peg j. Thus, if we define the predicate $\mathrm{P}(S, S')$ to be the relationship between the current state and the next states in terms of all permutations of locations for $\mathsf{move\_disk}(S, v_i, v_j, S')$, and define $\mathrm{Q}(S)$ to be the set of legal configurations of pegs on disks, then we can define the $\mathrm{T}(S)$ to be

$$T(S) = \{S' \ \mid \ S' \neq S \wedge \mathrm{P}(S, S') \wedge \mathrm{Q}(S') \}.$$

To complete our configuration of the problem-solving method for the disk-moving task, we must define (1) the number of pegs k, (2) the initial state S_I and goal state S_G (both states are represented as particular assignments of values to the k state variables, and both states may be run-time input), and (3) the predicate $\mathrm{Q}(S)$ that represents constraints on

permissible states.[4] In this model, our example towers-of-Hanoi problem (Figure 3) has the following definition:

1. $n = 3$.
2. $v_1(S_I) = (1, 2, \ldots, m-1, m)$, $v_2(S_I) = ()$, $v_3(S_I) = ()$,
   $v_1(S_G) = ()$, $v_2(S_G) = ()$, $v_3(S_G) = (1, 2, \ldots, n-1, n)$, where n = number of disks.
3. The states must satisfy the state-consistency constraint $\mathrm{Q}(S)$ that says that, for each state variable $v_i(S)$, if the value of $v_i(S)$ is $(d_{1,i}, d_{2,i}, \ldots, d_{k,i})$, then $d_{j,i} > d_{j+1,i}$.

Note that, with this definition of $\mathrm{Q}(S)$, we do not have to constrain the moves of the game with $\mathrm{P}(S, S')$. However, an alternative approach to this configuration of the problem-solving method is to use $\mathrm{P}(S, S')$ to constrain the use of the move_disk predicate such that no illegal moves will be performed. In this configuration, the constraint $\mathrm{Q}(S)$ becomes unnecessary.

4.1.2 The Board-Game Method for the Sisyphus Task

We can configure the board-game method to perform the Sisyphus room-assignment task. If we view the room-assignment task as a board game where the persons are pieces, and the rooms are locations, we can define legal moves for persons between rooms (or from the *unassigned* location). Initially, all persons are located outside the building (i.e., nobody is assigned a room). The goal is to bring all persons inside the building under the room-assignment constraints; the goal predicate checks for an empty outside location. Contradictions occur, for instance, when a smoker and a nonsmoker are assigned to the same room (because the problem definition stipulates that smokers and nonsmokers should not share rooms). If a contradiction develops, the problem solver backtracks, then attempts another series of moves. The result of this algorithm is the goal state in which persons have been assigned correctly to rooms.

The predicate possible_move(S, p, v_u, v_r) defines that it is legal to move the first unassigned person into any available room that matches the person's professional role. The state variable v_r, where $r \in \mathrm{R}$ and $\mathrm{R} = \{\text{C5-113}, \ldots, \text{C5-117}, \text{C5-119}, \ldots, \text{C5-123}\}$ (in the Sisyphus example; see Section 2.2), represents a room in the office building, and $v_r(S)$ is the set of persons assigned to v_r in state S. The definition of possible_move uses the function select$(v_u(S))$ to select the person to be assigned. The predicate unoccupied(S, v_r) is a help predicate that defines the situation in which a room can accommodate a person. The predicate contradiction(S) detects any contradiction where smokers and nonsmokers share the same room. In the initial state S_I, each person p in the set P of persons to assign is at the location *unassigned* ($v_u(S_I) = \mathrm{P}$). In the room-assignment task, there is no predefined goal state, because the final state is the result. We can define a goal predicate that tests whether the location representing *unassigned* is empty as $\forall S : \mathsf{goal}(S) \leftarrow v_u = \emptyset$. We define the possible_move and contradiction predicates for the room-assignment task as

$$\forall S, p, v_u, v_r, x_{\mathrm{role}} :$$

[4] Note that, however, for a generalized tower-of-Hanoi task, parts 1 and 2 can be run-time input to the problem solver. Also, it is conceivable that certain tasks will require the state consistency in part 3 to be input at run time.

$$
\begin{aligned}
&\mathsf{possible\_move}(S, p, v_u, v_r) \ \leftarrow \\
&\quad p \in \mathrm{P} \ \wedge \\
&\quad v_r \in \mathrm{R} \ \wedge \\
&\quad v_u(S) \neq \emptyset \ \wedge \\
&\quad p = \mathsf{select}(v_u(S)) \ \wedge \\
&\quad \mathsf{professional\_role}(p, x_{\mathrm{role}}) \ \wedge \\
&\quad (\mathsf{require\_large}(x_{\mathrm{role}}) \rightarrow \mathsf{large}(v_r)\,) \ \wedge \\
&\quad (\mathsf{require\_central}(x_{\mathrm{role}}) \rightarrow \mathsf{central}(v_r)\,) \ \wedge \\
&\quad \mathsf{unoccupied}(S, v_r),
\end{aligned}
$$

$$
\begin{aligned}
&\forall S, v_r, p, x_{\mathrm{role}} : \\
&\qquad \mathsf{unoccupied}(S, v_r) \ \leftarrow \\
&\qquad\quad v_r(S) = \emptyset \ \vee \\
&\qquad\quad (v_r(S) = \{p\} \wedge \mathsf{professional\_role}(p, x_{\mathrm{role}}) \wedge \mathsf{sharing}(x_{\mathrm{role}})\,),
\end{aligned}
$$

$$
\begin{aligned}
&\forall S, v_r, p_A, p_B : \\
&\qquad\qquad \mathsf{contradiction}(S) \ \leftarrow \\
&\qquad\qquad\quad v_r \in \mathrm{R} \ \wedge \\
&\qquad\qquad\quad p_A, p_B \in \mathrm{P} \ \wedge \\
&\qquad\qquad\quad p_A, p_B \in v_r(S) \ \wedge \\
&\qquad\qquad\quad \mathsf{smoker}(p_A) \wedge \neg\mathsf{smoker}(p_B).
\end{aligned}
$$

We do not claim that this configuration of the board-game problem leads to an exact solution to the Sisyphus room-assignment problem as defined in [25]. For the sake of brevity, and because we are merely using the room-assignment problem as a basis for our discussion on method selection and configuration, we have deliberately excluded from this method configuration certain aspects, such as the order in which persons are assigned to rooms, and constraints related to members of projects.

4.2 Specialization

In the definition of new methods, method designers can take advantage of methods created previously. By *specializing* methods to classes of tasks more narrow than those for which the methods were designed originally, designers can reuse much of the development work. Another view of method specialization is to regard the methods under design as tasks. We can implement the board-game method, for instance, by modeling the board-game *task* with a relatively general method. Figure 6 shows the specialization of chronological backtracking

to the board-game and propose-and-revise methods. In turn, we can specialize the propose-and-revise method to another instance of the board-game method (by *proposing* a board configuration and by *revising* the configuration by moving pieces).

What remains to be done at this point is to map the input and output of the configured method to the environment and to the domain ontology. Moreover, we must acquire the domain knowledge required by the method to perform its task. In the Section 5, we shall examine the relationships between problem-solving methods and ontologies; in Section 6, we shall discuss knowledge acquisition for methods.

5 Problem-Solving Methods and Ontologies

Developers cannot reuse problem-solving methods easily without considering the methods' input and output, as well as to the domain knowledge required by the method. The input that a method accepts and the output that the method generates must be defined such that the developer can map the task-level input and output to the method's input and output structures. We shall discuss the interaction between the methods and the declarative representations that model the domain. The artificial-intelligence community has adopted from metaphysics the term *ontology* for models that are concerned with the nature and relations of being [32; 37]. In the artificial-intelligence context, however, the term *ontology* usually denotes models that define *concepts* and *relationships* among concepts. These concepts can represent classes of material objects, abstract terms, artificially constructed classes, states of a system, and so on. In many aspects, ontologies are engineered artifacts that model the world for a particular purpose. Moreover, the accuracy and predictability of the models are relative to the task and the design of the system that uses them [6]. Frame systems and object-oriented programming languages provide an operational framework for defining and using ontologies. Most of these languages provide semantics for basic relations, such as *is-a* and *instance-of*. In PROTÉGÉ-II, we use ontologies to define the input and output of methods [17; 48].

Problem-solving methods and domain ontologies cannot be viewed in isolation. The design of a domain ontology affects how well methods can use the ontology for problem solving. Likewise, the method selected requires certain information for its problem-solving strategy, which affects the scope and organization of the domain ontology. For instance, a planning method might require the definition in the domain ontology of the *actions* relevant to the domain, and a classification method might require a taxonomy of domain-specific *hypotheses* for its problem-solving strategy. Development methodologies that incorporate method reuse must take into account this interdependence between ontologies and methods, as the development of knowledge-based systems is fundamentally an iterative process, which involves integrated modeling of declarative and procedural aspects of the application task. Bylander and Chandrasekaran [1] discuss this interaction problem in the context of knowledge acquisition for generic tasks. Linster [24] discusses the mapping between domain ontologies and problem-solving methods.

Problem-solving methods are designed to perform tasks that involve operations on complex data structures. In the PROTÉGÉ-II approach, we use the notion of *method ontologies* [46]. Method ontologies define the methods' interfaces to other methods and to other components of the application system (e.g., user and database interfaces). The method's input ontology defines the object structures that the method requires as input, and the output

ontology defines the output of the method. Figure 7 shows input and output ontologies for a problem-solving method, and their relationships to the *application ontology*. The developer must map the vocabulary defined by the application ontology to the method ontologies [17; 48]. In many cases, there is merely a *terminological* difference between definitions in the domain and method ontologies. For instance, when the board-game method is used for the Sisyphus room-assignment task, the method-specific terms *piece* and *location* correspond directly to the domain-specific terms *person* and *room*. The developer can accomplish such terminological mappings by linking corresponding concepts in the method and domain ontologies. However, sometimes there is a significant *semantic* difference between relevant concepts in the method and domain ontologies. One concept in the method ontology may correspond to several concepts in the domain ontology. For example, there is a significant semantic difference between the notion of *states* in the chronological-backtracking method and the concepts of *persons* and *rooms* in the Sisyphus task. In such cases, the developer must define the mapping in a language that is more expressive than is straightforward concept linking. Transformation rules are an example of an approach that allows the developer to define such complex mappings.

In addition to mapping the input and output of the problem-solving methods to domain ontologies, the developer must ensure that the appropriate domain knowledge is available to the methods. Problem-solving methods resemble miniature expert-system shells in that they are designed to perform a task by drawing conclusions from a knowledge base. Such *method knowledge bases* contain the domain knowledge that the method requires to perform the task. For example, a classification method might require a set of classification rules, and a planning method might require a set of preconditions (e.g., for actions), the members of which are expressed as rules. Typically, methods invoke the method knowledge bases at certain points in the problem-solving strategy, such as when they must make a complex decision, and when they must derive a value. McDermott [29] refers to these inferences from a knowledge base as knowledge *roles*. Figure 8 illustrates how a problem-solving method uses its knowledge base. In Section 6, we shall discuss how the developer can approach the knowledge-acquisition problem for the method knowledge bases.

6 Knowledge Acquisition

Domain knowledge can be acquired conveniently by a method-specific knowledge-acquisition tool that allows experts to enter, review, and edit domain knowledge [29]. One approach to providing support in the form of a knowledge-acquisition tool is to associate a generic knowledge-acquisition tool with each problem-solving method [21; 28; 41]. Our approach, however, is to generate and custom tailor a knowledge-acquisition tool independent of the problem-solving methods that are part of the design for the knowledge-based system [8; 9; 11; 12; 13; 38]. Our motivation for generating knowledge-acquisition tools independent of the problem-solving methods is that the cognitive basis for the (declarative) domain knowledge that the knowledge base models is different from the cognitive basis for the operations that the problem-solving method performs.

Analogous to the task analysis, the development of a domain-oriented knowledge-acquisition tool must be preceded by a *knowledge-acquisition analysis*. The purpose of knowledge-acquisition analysis is to examine the development situation from a knowledge-acquisition point of view, and to formulate requirements for tool support. This analysis involves (1) iden-

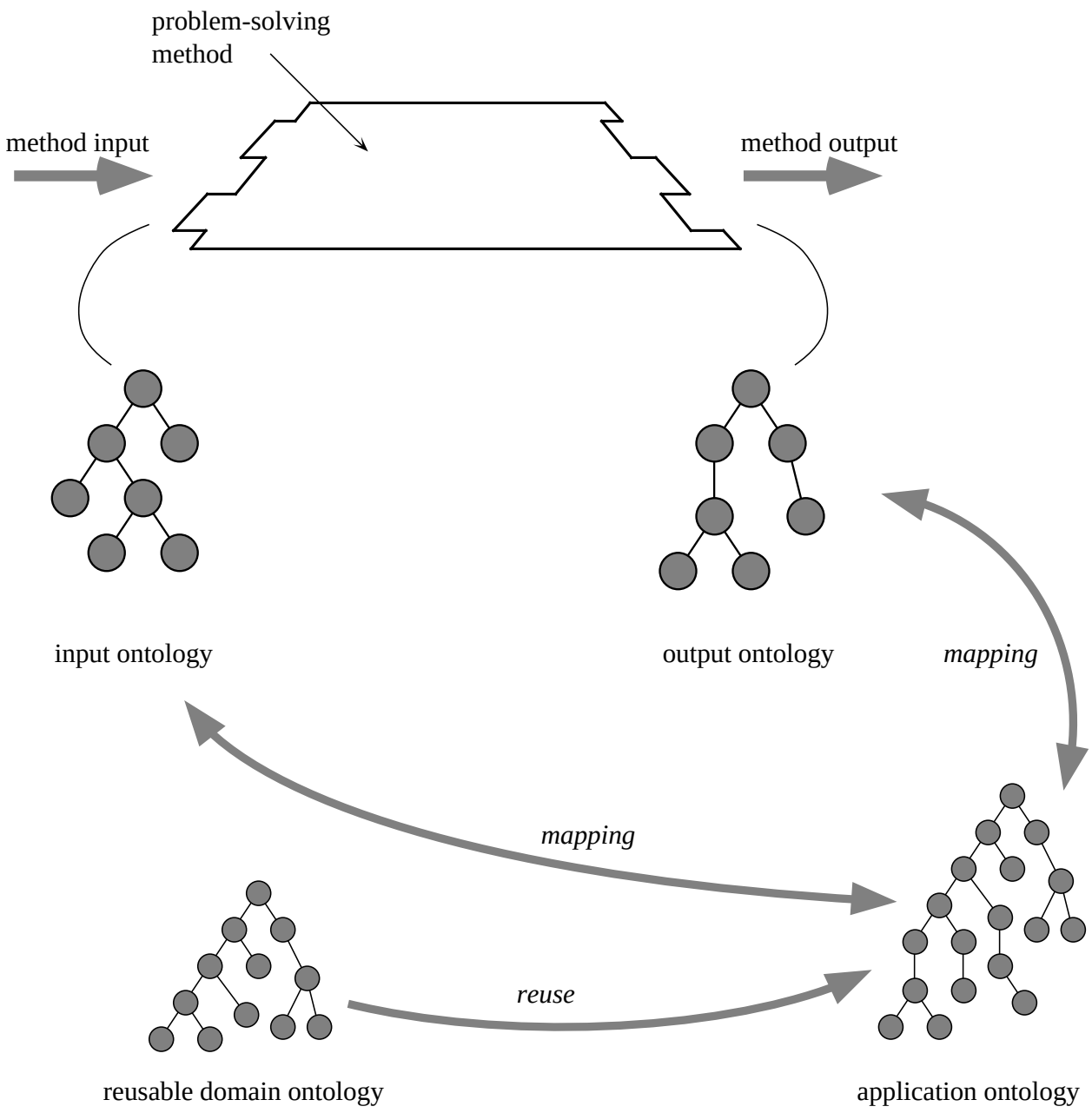

Figure 7: Method ontologies. The *input* and *output* ontologies define the input and output of the method. In PROTÉGÉ-II, the developer defines mappings between the input and output ontologies and the application ontology. The developer can design the application ontology by reusing parts of domain ontologies (which can be applicable to several applications).

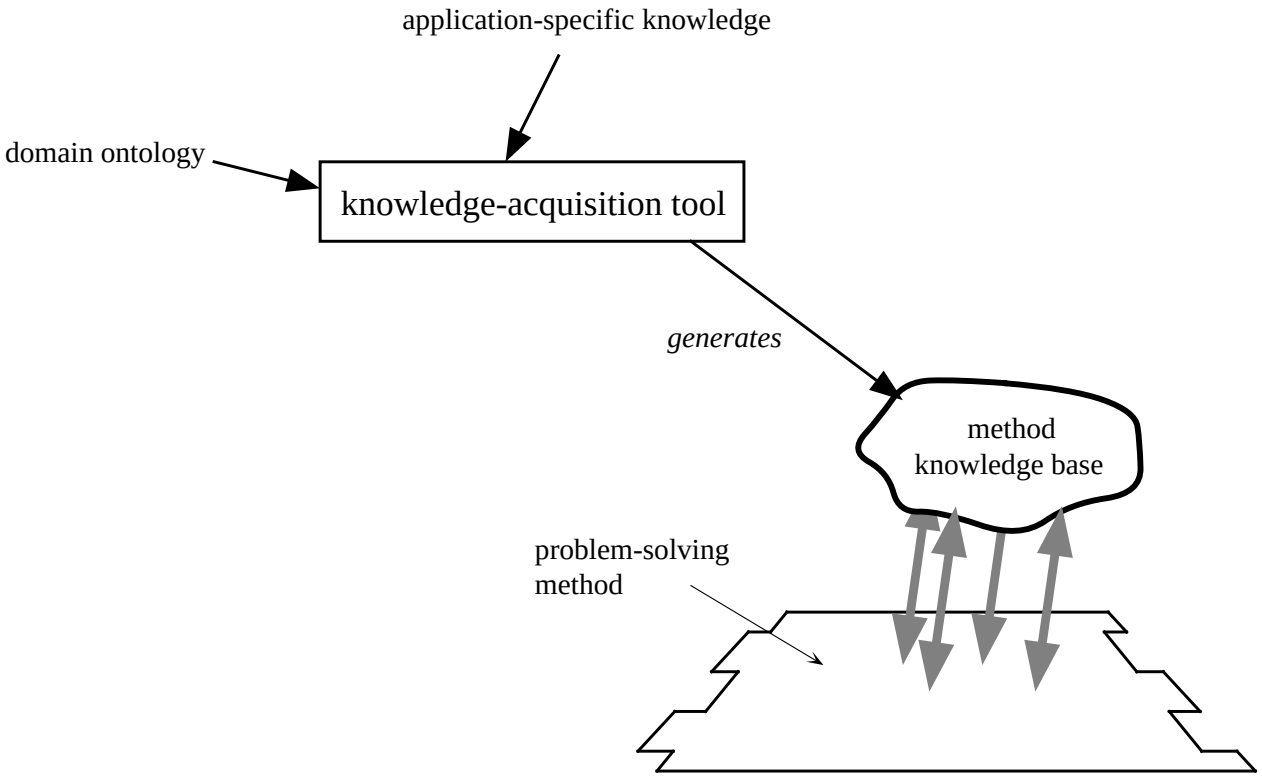

Figure 8: Method knowledge bases. To make inferences during problem solving, methods can use knowledge bases that define domain knowledge, but are specific and local to each method. A knowledge-acquisition tool may optionally generate these domain- and method-specific knowledge bases. In PROTÉGÉ-II, the developer defines a mapping from the output of the knowledge-acquisition tool (which consists of instances of application-ontology classes) to the method knowledge base (which consists of instances of method-ontology classes) [17].

tification of the users of the knowledge-acquisition tool (e.g., domain experts); (2) segmentation of the part of the knowledge base that is to be acquired through the tool; (3) definition of a language in which to express the knowledge (e.g., a graphical language); and (4) specification of semantics for this language, as well as of denotational semantics that describe the generation of knowledge bases. Knowledge-acquisition analysis is thus a phase in the design of support tools. Alternatively, knowledge-acquisition analysis requires selection of a preexisting tool.

When the knowledge-acquisition situation has been analyzed and the role for the knowledge-acquisition tool has been established, the developer can design the tool. If we examine the Sisyphus room-assignment problem and the sample transcript provided, we find that the expert is concerned not so much with the individuals themselves, as with their *professional roles* in the group. The transcript contains statements such as: "The head of the group needs a central room." (The fact that a particular person is the director of the group must then be described in the run-time input data.) Another important observation is that the professional roles provide a specification, or sometimes a justification, for the type of room to which a person should be assigned—for instance, the head of the group requires a large single room, whereas a staff researcher may share a room with another person. Hence, our hypothesis is that much of the domain knowledge required for room assignment can be expressed in the form of rules, where the premise matches a certain professional role, and where the rule conclusion is a room specification (e.g., a query to a database of rooms available). We are primarily interested in expressing a mapping from professional roles to potential assignments of persons to rooms. An example of a rule in this rule set follows:

$$\forall(p, r)\ (p \in \text{person} \wedge r \in \text{professional\_role})\text{: head-of-group}(p) \rightarrow$$
$$\text{require\_large}(r) \wedge \text{require\_central}(r).$$

To illustrate how a knowledge-acquisition tool that is custom tailored for the Sisyphus room-assignment task can be designed according to this analysis, we have implemented a prototype tool in the metatool DASH [13]. DASH is a component of the PROTÉGÉ-II architecture. DASH takes as input a domain ontology, and produces as output a knowledge-acquisition tool that allows domain specialists to create instances of classes in that ontology. DASH supports the developer in creating a dialog structure for the knowledge-acquisition tool, and in designing layouts for form-based knowledge editors. Figure 9 shows a sample screen from the generated knowledge-acquisition tool that allows the expert to specify new professional roles for the Sisyphus problem solver. The knowledge-acquisition tool in Figure 9 produces CLIPS frame instances from the entries that users make into these forms. The CLIPS implementation of the problem solver is configured to use these instances to select an appropriate room for each professional role.

Up to this point, we have mainly examined reusable problem-solving methods from an abstract view—that is, we have not concerned ourselves with the actual implementation of methods within a programming language. In Section 7, we shall continue with a discussion of results and lessons learned from an implementation of the methods and tasks that we discussed in the previous sections.

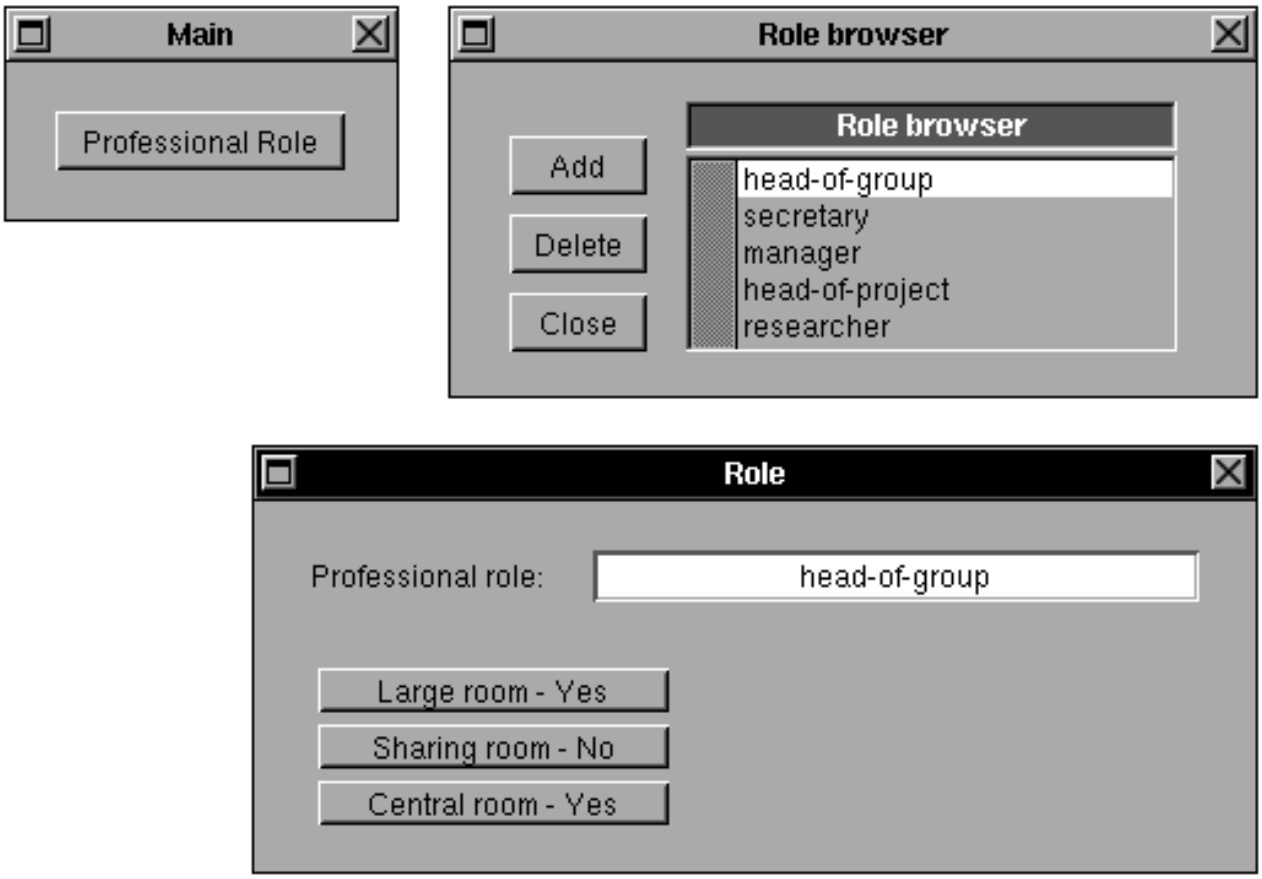

Figure 9: A screen display from the knowledge-acquisition tool for the Sisyphus room-assignment problem. The main menu (upper left) provides access to a professional-role browser (upper right), which can be used to open five different professional-role forms (lower right).

Table 4: Lines of CLIPS code *added* to the chronological-backtracking and board-game methods to configure them for various tasks.

Task	Problem-solving method	Lines of code
towers of Hanoi	chronological backtracking	76
farmer's dilemma	chronological backtracking	72
Sisyphus room assignment	chronological backtracking	278
towers of Hanoi	board game	17
farmer's dilemma	board game	31
cannibals and missionaries	board game	32
Sisyphus room assignment	board game	39

7 Implementation Results

Probably the most important factor that determines the utility of reusable problem-solving methods is the time required to configure a method for a particular task and to integrate that method with other methods in the knowledge-based system under development. We seek to develop a framework for configuration of problem-solving methods that minimizes the knowledge-engineering time required. Because method configuration in first-order logic may not reflect accurately method reuse and configuration in practical development (where existing programming languages are used as the implementation vehicle), we shall examine the utility of method reuse in CLIPS. Table 4 shows the numbers of lines of CLIPS code that were required to configure the chronological-backtracking and board-game methods for various tasks (i.e., by providing T-functions and move rules, respectively). Although there are many problems associated with measuring the complexity of a program by counting the number of program lines, the number of lines is one of the simplest and most intuitive measures we have. Moreover, because the programs analyzed are relatively small, the differences among various software metrics are minor for our purposes. Although it is difficult to measure the design and implementation effort objectively, we believe that the lines of code correlate well to the implementation effort in this project.

As shown in Table 4, the *configuration* of the chronological-backtracking method for the Sisyphus room-assignment problem required 278 lines of code. The chronological-backtracking method itself, however, was implemented using a recursive algorithm in six lines of code (without utility functions for management of various data structures). In a sense, the chronological-backtracking method is highly reusable, because it is general and simple. However, this method requires substantial work if it is to be reused for any interesting task.

The board-game method, however, allows for much more compact method configurations than does chronological backtracking. The *configuration* of the board-game method for the Sisyphus task required only 39 lines of code. The results for the other problems we tried are similar (see Table 4). The level of reuse (at least, in terms of additional program lines) is significantly larger when these problems are implemented with the board-game method. Figure 10 illustrates the proportion of reuse in the configurations of the chronological-backtracking and board-game methods for the Sisyphus room-assignment task. The implementation of the chronological-backtracking method required 35 lines of utility functions (e.g., list-manipulation functions) and six lines of recursive definition of chronological backtracking.

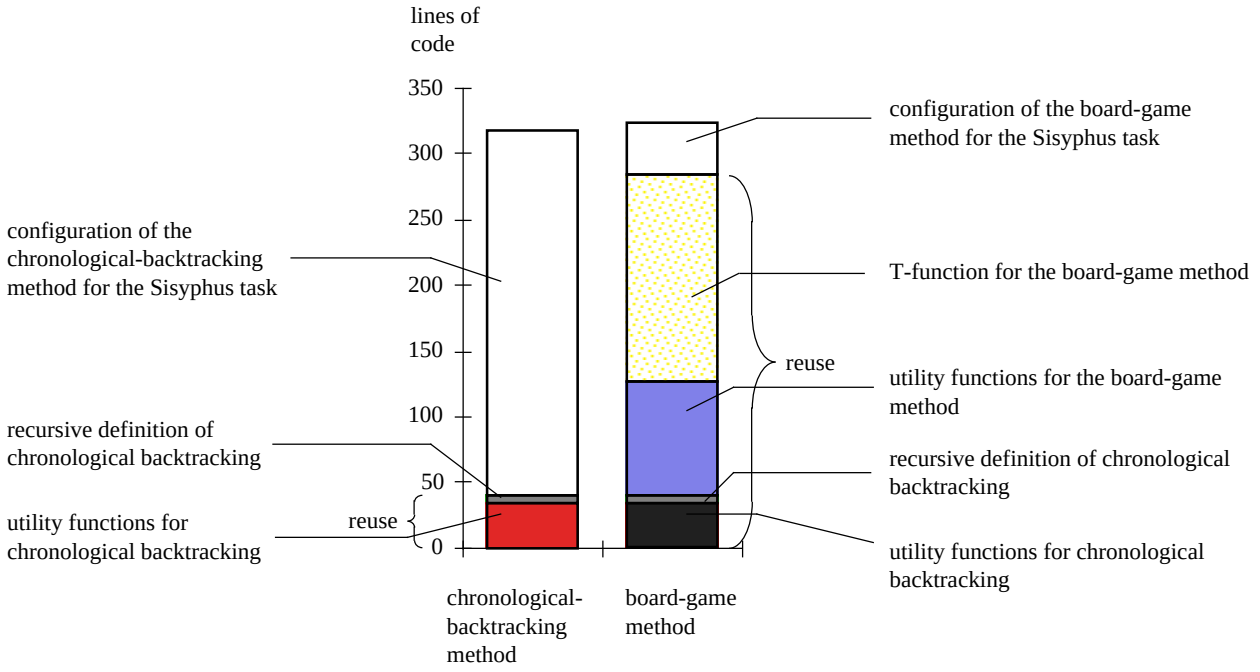

Figure 10: The disposition of CLIPS code in the configuration of the chronological-backtracking and board-game methods for the Sisyphus room-assignment task. The columns represent the chronological-backtracking and board-game methods, respectively. Note that the total amount of CLIPS code required to implement a program that accomplishes the room-assignment task is approximately the same for the chronological-backtracking and board-game methods (319 versus 325 lines of code). However, the board-game method allows us to *reuse* more code than does the chronological-backtracking method.

The configuration of the chronological-backtracking method for the Sisyphus task required 278 lines. In this case, 13 percent of the complete program for the Sisyphus problem consists of reused code. The implementation of the board-game method is based on the chronological-backtracking method, and includes an additional 78 lines for utility functions and 158 lines for the T-function that defines the board-game method in terms of chronological backtracking.[5] The configuration of the board-game method required 39 lines. In the board-game case, 88 percent of the complete program for the Sisyphus problem consists of reused code.

8 Related Work

Several research groups are developing architectures for reusable problem-solving methods. We shall discuss four important approaches that are related closely to the PROTÉGÉ-II framework.

Chandrasekaran [2; 3] was among the first researchers to suggest the development of knowledge-based systems from reusable components, or *generic tasks*. A generic task defines both a class of application tasks with common features, and a method for accomplishing

[5] The board-game method reuses code from the chronological-backtracking method.

these tasks. In his more recent work on task analysis, Chandrasekaran [4] uses *task structures* for modeling of application tasks. Task structures lay out the relationships between a task, the problem-solving methods for it, the knowledge requirement for the methods, and the subtasks that the methods set up. In this approach, as in PROTÉGÉ-II, the developer models new tasks by identifying appropriate task structures recursively. Early versions of the generic-task approach used problem-solving methods of a relatively large grain size (e.g., the order of planners and schedulers) when compared to PROTÉGÉ-II.

In the *components-of-expertise* approach [44], Steels and his colleagues are developing *application kits* that provide a collection of software artifacts that developers can use to build a knowledge-based system [45]. Application kits contain most of the components required to develop target systems that perform a certain class of application tasks (e.g., configuration and planning). At the highest level, this approach is based in three perspectives of the target system: *models* (domain ontologies), *methods,* and *tasks.* The developer refines these perspectives successively in a spiral-development approach that moves toward the execution and code levels of the target system. An important feature of the application-kit approach is that the application kits are built from primitive elements that the developer can inspect and modify. Steels and his colleagues are developing the KREST workbench, which supports the development of knowledge-based systems from application kits.

An important difference between PROTÉGÉ-II and the application-kit approach is that the goal of PROTÉGÉ-II is to minimize the programming required for method reuse, whereas KREST defines models, methods, and tasks at several architectural levels, including the code level. Like PROTÉGÉ-II, KREST allows the developer to combine explicitly domain ontologies and problem-solving methods to instantiate models for application tasks. KREST, however, requires that models be defined in terms of Lisp data structures and that problem-solving methods be defined as Lisp program code. Another significant difference between PROTÉGÉ-II and the approaches of Chandrasekaran and Steels is that an important goal of the PROTÉGÉ-II environment is to support the generation of domain-specific knowledge-acquisition tools.

Spark, Burn, and FireFighter (SBF) [21; 28] constitute a set of tools that is designed to help nonprogrammers and developers to build application programs. The SBF approach relies heavily on *workplace analysis* for modeling of the application task. Spark is a configuration tool that allows the developer to build problem solvers from reusable nondecomposable components that, in the SBF framework, are called *mechanisms.* The grain size of such mechanisms can be up to whole application programs. Each mechanism has associated with it a knowledge-acquisition tool that elicits and generates the knowledge required by the mechanism to perform the latter's task. Burn is a development tool that elicits domain knowledge from application specialists by invoking mechanism-specific knowledge-acquisition tools. FireFighter is a debugging tool that helps the developer to debug the final application system.

The DIDS [41] framework for development of knowledge-based systems also uses reusable *mechanisms* as its basic components. DIDS is designed for the modeling of configuration-design tasks. Such tasks involve the construction of a design (e.g., a design of an elevator) based on a fixed set of parts. In the DIDS approach, the target systems select parts, and interconnect them according to design specifications provided by the end users. The DIDS library of mechanisms allows the developer to construct new problem-solving methods for new design tasks. The mechanisms operate on a standardized knowledge representation. Also, the DIDS framework supports the configuration of method-specific knowledge-acquisition tools for the acquisition of relevant configuration knowledge. The SBF and DIDS frameworks are similar

to that of PROTÉGÉ-II in that all approaches emphasize generation of knowledge-acquisition tools. However, the PROTÉGÉ-II approach to generation of knowledge-acquisition tools differs from that of SBF and DIDS in that PROTÉGÉ-II uses domain ontologies as the basis for the tool generation to create a coherent dialog structure for the target tool.

9 Summary and Conclusions

Reusable problem-solving methods provide building blocks for developers of knowledge-based systems. In essence, such methods are abstractions of problem-solving behavior that capture procedural knowledge for accomplishing a task. We have studied the *selection* and *configuration* of methods for several different tasks, and have described how the input-and-output requirements of problem-solving methods must be mapped onto domain ontologies. In addition, we have discussed two supplementary design activities: task analysis and knowledge acquisition. The *chronological-backtracking* and *board-game* methods served as the basis for our examination of task modeling with reusable problem-solving methods.

An important question for all reusable problem-solving methods concerns the scope for the methods. Researchers distinguish between *general* and *role-limiting* methods for problem solving.[6] The disadvantage of general methods is that they do not constitute knowledge roles that can guide sufficiently the method configuration and the acquisition of the knowledge required for the method [29]. Role-limiting methods, on the other hand, provide more structure and guidance for method configuration than do general methods. Not surprisingly, the results of our work confirm that we can indeed use both general and role-limiting methods to model many tasks, but that the cost of using general methods might be too high, especially when the amount of reused code is taken into account. More important, we showed how method designers can define new methods by making additional ontological commitments to preexisting methods, and by specializing their behavior. Such ontological commitments to general methods can result in more specific methods that decrease significantly the work required for task modeling. For example, the ontological commitments made by the board-game method help developers to map new tasks to the method, and to configure the method for new tasks. The work required to configure the board-game method for the Sisyphus room-assignment task is significantly less than that required to configure the chronological-backtracking for the same task.

The modeling support of a problem-solving method is determined by the context in which that method is used. One of the most important factors for the reusability of a method is the *cognitive distance* between the method and the task that the developer models with the method. Moreover, problem-solving methods must support conceptual models that make the methods explainable to developers, and intuitive to reuse. The notions of general and role-limiting methods are de facto context dependent, and are relative to the tasks being modeled. Methods that provide clear mental models for problem solving help method designers to communicate results, and help developers to understand how methods operate, and how methods can be configured to perform new tasks. Given a library of such methods, the developer can select an appropriate method, configure it to perform particular application

[6] Sometimes, the phrases *weak method* and *strong method* are used to denote general and specific methods, respectively. Weak methods make only weak assumptions about the task (i.e., a general method such as chronological backtracking). Strong methods make strong assumptions about the task (i.e., a specific method, such as the board-game method).

tasks, and, optionally, generate a knowledge-acquisition tool that elicits the domain knowledge required for problem solving.

Acknowledgments

This work has been supported in part by grants LM05157 and LM05208 from the National Library of Medicine, by grant HS06330 from the Agency for Health Care Policy and Research, by gifts from Digital Equipment Corporation, and by scholarships from the Swedish Institute, from the Fulbright Commission, and from Stanford University. Dr. Musen is recipient of National Science Foundation Young Investigator Award IRI-9257578.

We are grateful to Lyn Dupré for providing editorial assistance. We thank John Gennari, Marc Linster, Thomas Rothenfluh, and Eckart Walther for their comments on a previous draft of this article.

References

[1] Tom Bylander and B. Chandrasekaran. Generic tasks for knowledge-based reasoning: The "right" level of abstraction for knowledge acquisition. *International Journal of Man–Machine Studies*, 26(2):231–243, 1987.

[2] B. Chandrasekaran. Towards a taxonomy of problem-solving types. *AI Magazine*, 4(1):9–17, 1983.

[3] B. Chandrasekaran. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, 1(3):23–30, 1986.

[4] B. Chandrasekaran. Design problem solving: A task analysis. *AI Magazine*, 11(4):59–71, 1990.

[5] William J. Clancey. Heuristic classification. *Artificial Intelligence*, 27(3):289–350, 1985.

[6] William J. Clancey. The frame of reference problem in the design of intelligent machines. In K. Van Lehn, editor, *Architectures for Intelligence: The 22nd Carnegie–Mellon Symposium on Cognition*, chapter 13, pages 357–423. Lawrence Earlbaum Associates, Hillsdale, NJ, 1991.

[7] Thomas Dean. Planning paradigms, in William Swartout, editor, DARPA Santa Cruz workshop on planning. *AI Magazine*, 9(2):115–119, Summer 1988.

[8] Henrik Eriksson. *Meta-Tool Support for Knowledge Acquisition.* PhD thesis 244, Linköping University, Linköping, Sweden, 1991.

[9] Henrik Eriksson. Metatool support for custom-tailored domain-oriented knowledge acquisition. *Knowledge Acquisition*, 4(4):445–476, 1992.

[10] Henrik Eriksson. Specification and generation of custom-tailored knowledge-acquisition tools. In *Proceedings of the Thirteen International Joint Conference on Artificial Intelligence, IJCAI'93*, pages 510–515, Chambéry, Savoie, France, August 29 – September 3 1993.

[11] Henrik Eriksson and Mark A. Musen. Conceptual models for automatic generation of knowledge-acquisition tools. *Knowledge Engineering Review*, 8(1):27–47, 1993.

[12] Henrik Eriksson and Mark A. Musen. Metatools for knowledge acquisition. *IEEE Software*, 10(3):23–29, May 1993.

[13] Henrik Eriksson, Angel R. Puerta, and Mark A. Musen. Generation of knowledge-acquisition tools from domain ontologies. In *Proceedings of the Eighth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, January 1994.

[14] Larry Eshelman, Damien Ehret, John McDermott, and Ming Tan. MOLE: A tenacious knowledge-acquisition tool. *International Journal of Man–Machine Studies*, 26(1):41–54, 1987.

[15] D. Fensel, J. Angele, and D. Landes. KARL: A knowledge acquisition and representation language. In *Proceedings of Expert Systems and their Applications, 11th International Workshop, Conference Tools, Techniques & Methods*, pages 513–528, Avignon, France, 1991.

[16] R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of AAAI-87*, pages 202–206, 1987.

[17] John H. Gennari, Samson W. Tu, Thomas E. Rothenfluh, and Mark A. Musen. Mapping domains to methods in support of reuse. In *Proceedings of the Eighth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 24.1–24.20, Banff, Canada, January 1994.

[18] Roger Hale. Temporal logic programming. In Antony Galton, editor, *Temporal Logics and Their Applications*, chapter 3, pages 91–119. Academic Press, London, 1987.

[19] Frank van Harmelen and John Balder. $(ML)^2$: A formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1):127–161, 1992.

[20] Werner Karbach, Marc Linster, and Angi Voß. Models, methods, roles and tasks: Many labels—one idea? *Knowledge Acquisition*, 2(4):279–299, 1990.

[21] Georg Klinker, Carlos Bhola, Geoffroy Dallemagne, David Marques, and John McDermott. Usable and reusable programming constructs. *Knowledge Acquisition*, 3(2):117–135, 1991.

[22] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[23] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[24] Marc Linster. Linking modeling to make sense and modeling to implement systems in an operational modeling environment. In Th. Wetter, K.-D. Althoff, J. Boose, B.R. Gaines, M. Linster, and F. Schmalhofer, editors, *Current Developments in Knowledge Acquisition: EKAW'92*, pages 55–74. Springer-Verlag, Berlin, Germany, 1992.

[25] Marc Linster, editor. *Sisyphus'92: Models of Problem Solving*, Technical Report 630, Gesellschaft für Mathematik und Datenverarbeitung (GMD), St. Augustin, Germany, 1992.

[26] Sandra Marcus and John McDermott. SALT: A knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence*, 39(1):1–37, 1989.

[27] Sandra Marcus, Jeffrey Stout, and John McDermott. VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, 9(1):95–112, Spring 1988.

[28] David Marques, Geoffroy Dallemange, Georg Klinker, John McDermott, and David Tung. Easy programming: Empowering people to build their own applications. *IEEE Expert*, 7(3):16–29, June 1992.

[29] John McDermott. Preliminary steps toward a taxonomy of problem-solving methods. In Sandra Marcus, editor, *Automating Knowledge Acquisition for Expert Systems*, chapter 8, pages 225–256. Kluwer Academic Publishers, Boston, MA, 1988.

[30] Mark A. Musen. *Automated Generation of Model-Based Knowledge-Acquisition Tools*. Morgan-Kaufmann, San Mateo, CA, 1989.

[31] Mark A. Musen. Automated support for building and extending expert models. *Machine Learning*, 4:349–377, 1989.

[32] Mark A. Musen. Dimensions of knowledge sharing and reuse. *Computers and Biomedical Research*, 25:435–467, 1992.

[33] Mark A. Musen. Overcoming the limitations of role-limiting methods. *Knowledge Acquisition*, 4(2):165–170, 1992.

[34] Mark A. Musen, Lawrence M. Fagan, David M. Combs, and Edward H. Shortliffe. Use of a domain model to drive an interactive knowledge-editing tool. *International Journal of Man–Machine Studies*, 26(1):105–121, 1987.

[35] Mark A. Musen and Samson W. Tu. Problem-solving models for generation of task-specific knowledge-acquisition tools. In J. Cuena, editor, *Knowledge-Oriented Software Design*, pages 23–50. Elsevier, Amsterdam, 1993.

[36] NASA. *CLIPS Reference Manual*. Software Technology Branch, Lyndon B. Johnson Space Center, NASA, Houston, TX, 1991.

[37] R. Neches, R. Fikes, T. Finin, T. Gruber, T. Senator, and W.R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, Fall 1991.

[38] Angel R. Puerta, John W. Egar, Samson W. Tu, and Mark A. Musen. A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. *Knowledge Acquisition*, 4(2):171–196, 1992.

[39] Angel R. Puerta, Samson W. Tu, and Mark A. Musen. Modeling tasks with mechanisms. *International Journal of Intelligent Systems*, 8(1):129–152, 1993.

[40] Thomas E. Rothenfluh, John H. Gennari, Henrik Eriksson, Angel R. Puerta, Samson W. Tu, and Mark A. Musen. Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTÉGÉ-II solutions to Sisyphus-2. In *Proceedings of the Eighth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 43.1–43.30, Banff, Canada, January 1994.

[41] Jay T. Runkel and William P. Birmingham. Knowledge acquisition in the small. *Knowledge Acquisition*, 5(2):117–243, 1993.

[42] Yuval Shahar and Mark A. Musen. RÉSUMÉ: A temporal-abstraction system for patient monotoring. *Computers and Biomedical Research*, 26:255–273, 1993.

[43] Yuval Shahar, Samson W. Tu, and Mark A. Musen. Knowledge acquisition for temporal-abstraction mechanisms. *Knowledge Acquisition*, 4(2):217–236, 1992.

[44] Luc Steels. Components of expertise. *AI Magazine*, 11(2):28–49, Summer 1990.

[45] Luc Steels. Reusability and knowledge sharing. In Luc Steels and B. Lepape, editors, *Enhancing the Knowledge Engineering Process: Contributions from ESPRIT*, pages 240–270. Elsevier Publishers, Amsterdam, 1992.

[46] Samson W. Tu, Henrik Eriksson, John H. Gennari, Yuval Shahar, and Mark A. Musen. Ontology-based configuration of problem-solving methods and generation of knowledge-acquisition tools: Application of PROTÉGÉ-II to protocol-based decision support. *Artificial Intelligence in Medicine*, in press.

[47] Samson W. Tu, Michael G. Kahn, Mark A. Musen, Jay C. Ferguson, Edward H. Shortliffe, and Lawrence M. Fagan. Episodic skeletal-plan refinement based on temporal data. *Communications of the ACM*, 32(12):1439–1455, 1989.

[48] Eckart Walther, Henrik Eriksson, and Mark A. Musen. Plug-and-play: Construction of task-specific expert-system shells using sharable context ontologies. In *Proceedings of the AAAI Workshop on Knowledge Representation Aspects of Knowledge Acquisition*, San Jose, CA, July 1992.

[49] B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5–53, 1992.

Appendix

A Methods for the Towers of Hanoi Task

For the towers-of-Hanoi task, we shall consider five problem-solving methods. The first method, state-space search by chronological backtracking (Appendix A.1), is general and can be configured to accomplish many tasks, whereas the other methods discussed (Appendices A.2–A.5) make further commitments to the structure of the towers-of-Hanoi task, and take advantage of specific domain insights.

A.1 Chronological-Backtracking Method

State-space search by chronological backtracking explores the space of admissible states (see Figure 11). The method searches for a sequence of states, where the first state is a given *initial* state, the final state is a given *goal* state, and two consecutive states in the sequence satisfy constraints on how states can follow one another. The method ontology consists of a definition of *problem states* (S_i) and an *equality predicate* defined between any two states. The method uses the equality predicate to avoid circularities among states, and to identify the final state. There are three *inputs* to the method: (1) an initial state S_I, (2) a goal state S_G, and (3) constraints on the legal states $\mathrm{C}(S_i)$. The constraints on the legal states are defined as part of the method configuration (Section 4.1.1). The *output* of the method is a list of states $(S_1, \ldots, S_k)$ that satisfies the following conditions:

1. $S_1 = S_I$.
2. $S_k = S_G$.
3. For each i such that $1 \leq i \leq k$, S_i satisfies the legal-state constraints $\mathrm{C}(S_i)$; and, for each i, $1 \leq i < k-1$, S_{i+1} is a member of $\mathrm{T}(S_i)$, where $\mathrm{T}(S_i)$ is the output of a subtask that generates subsequent states, given input S_i.

To reuse chronological backtracking as a method, the developer must define how permissible next states are produced from any given state. The chronological-backtracking method uses a generator for subsequent states, which we shall call *transition function (T-function)*. An optional *sorting function (S-function)* reorders the output of the T-function. This sorting function encodes domain heuristics that reorder the set of possible next states, such that states that are likely to be on the solution path are tried first. The method has two *subtasks* (in the PROTÉGÉ-II sense): the *next-states* subtask (the T-function, $\mathrm{T}(S_i)$) and an optional *sort-states* subtask (the S-function). The next-state subtask requires, as inputs, a state S_i, and a predicate $\mathrm{P}(S_i, S_j)$ that, given S_i, constrains the legal next state. The output of the subtask is a set of states S_j that satisfies the predicate $\mathrm{P}(S_i, S_j)$. The input of the sort-states subtask is a list of states, and the output is a reordered list of the states on the input list. The algorithm of the chronological-backtracking method can be expressed as follows:

1. Define two internal variables X and Y, such that X keeps track of all states that the algorithm has considered already, and Y is a list of possible next states. Start the algorithm by setting Y to the list consisting of the initial state S_I.
2. If Y is the empty list, then stop, because there is no solution.

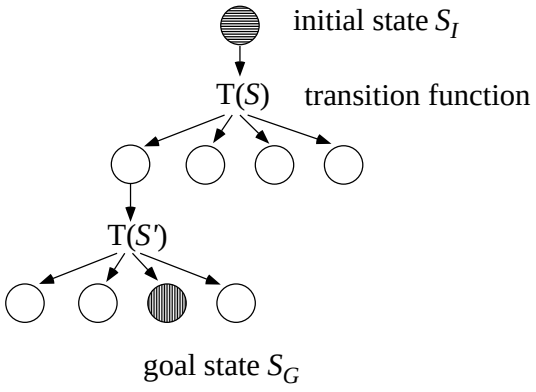

Figure 11: State-space search by chronological backtracking. This method explores the space of game states. The transition function T(S) is responsible for generation of subsequent states from the current state.

3. If the first state of Y is the goal state S_G, then add the state at the end of the X variable, return the value of the X as the solution, and stop.

4. If the first state of Y is a member of X, then delete the first state from Y, and go to step 2.

5. If the first state of Y is not a member of X, then perform the subtask *next-states* with the first state of Y as input, and go to step 2.

Chronological backtracking is a method that developers can configure (by providing a T-function) to solve most problems that can be accomplished by search. The work of modeling an arbitrary task as a T-function, however, might be extensive. In the worst case, assuming that no lexicographic ordering of the state production is possible, the method would need O(k^n) time (see Appendix A.6).

A.2 Recursive Task Decomposition

The method of *recursive task decomposition* (RTD) decomposes the overall task into subtasks that can be accomplished by a basic method. In general, to specify a solution by RTD, we need a base case and a specification of the recursion in terms of the input task and of that task's decomposition into simpler tasks of the same nature. For instance, we can decompose the classic towers-of-Hanoi task in the following way, to move a tower of n disks from peg A to peg B, using peg C:

```
classic_transfer(n, A, B, C) =
   {if (n = 0) then
       return
    else
      {classic_transfer(n-1, A, C, B);
       move(1, A, B);
       classic_transfer(n-1, C, B, A)}
   }
```

where `move(1, A, B)` is the basic move operation for the towers-of-Hanoi board game. The operation `move(1, A, B)` moves the top disk from peg A to peg B. The use of the RTD method, however, requires considerable domain and task-specific knowledge to suggest a viable decomposition leading to a correct solution. Note that the specific RTD solution outlined could work for any tower configuration of the towers-of-Hanoi problem, and for any value of n or k (without using the additional pegs); for only the classic version, however, does it produce the optimal solution (because it cannot take advantage of additional pegs). Notice that there is an implied overhead of $O(n)$ space just to maintain the stack of tasks.

A.3 Iterative Method

In an *iterative* method, the developer must provide a set of rules, or an algorithm, that specifies a definitive transformation from one game state to another, starting with the initial state and ending with the goal state. One such instantiation of the iterative method is the following simple set of rules, where the main idea is derived from a *topological* representation of the board [18]. Represent the pegs as a cycle (in the classic case, the triangle A, B, C). The smallest of the n disks moves around the cycle: clockwise if n is odd, and counterclockwise if n is even. After each move of the smallest disk, the only other legal move (by the current second-largest movable disk) is made. Figure 12 illustrates the use of these rules for $n = 3$ and for the triangle A, B, C of pegs. Note that the iterative method supplies a new, additional quality to the solution: It is *executable* in the sense that we do not, strictly speaking, produce a plan (unless we save a list of moves); rather, we obtain an execution trace of the optimal solution. The iterative solution in this form is specific to towers-of-Hanoi configurations of the tower type, and is optimal for only the classic towers-of-Hanoi task.

A.4 Piece-Oriented Method

The *piece-oriented* method is an object-oriented version of the iterative method. In this approach, each disk can determine when it should move. In the piece-oriented method, the developer must provide a uniform set of rules that describe when a piece should move. The state of the game either is not used, or is used in a limited fashion. The pegs are represented as a cycle. Disk i, $i = 0, 1, \ldots, n-1$, moves one location to the left or to the right each time in the same direction, depending on the parity of $(n - i)$: clockwise if it is odd, and counterclockwise if it is even. The following set of rules is adapted from a temporal analysis of the towers-of-Hanoi problem [18], and uses both a topological and a temporal representation:

1. The (discrete) time units are numbered $(0, 1, \ldots, 2^{n-1})$.
2. Initially, $\forall i$, disk i makes its first move at time unit 2^i.
3. Subsequently, $\forall i$, disk i moves, every 2^{i+1} time units (after its first move).

If we examine the example of the iterative method closely (see Figure 12), we can see that the smallest disk ($i = 0$) moves clockwise, whereas the second smallest disk ($i = 1$) moves counterclockwise. A disk moves independently at time points where its move condition is fulfilled. The concurrent-processing solution, in this form, is optimal only when applied to the classic towers-of-Hanoi task; it can be used for any number of pegs, but, unless generalized, it will not be optimal.

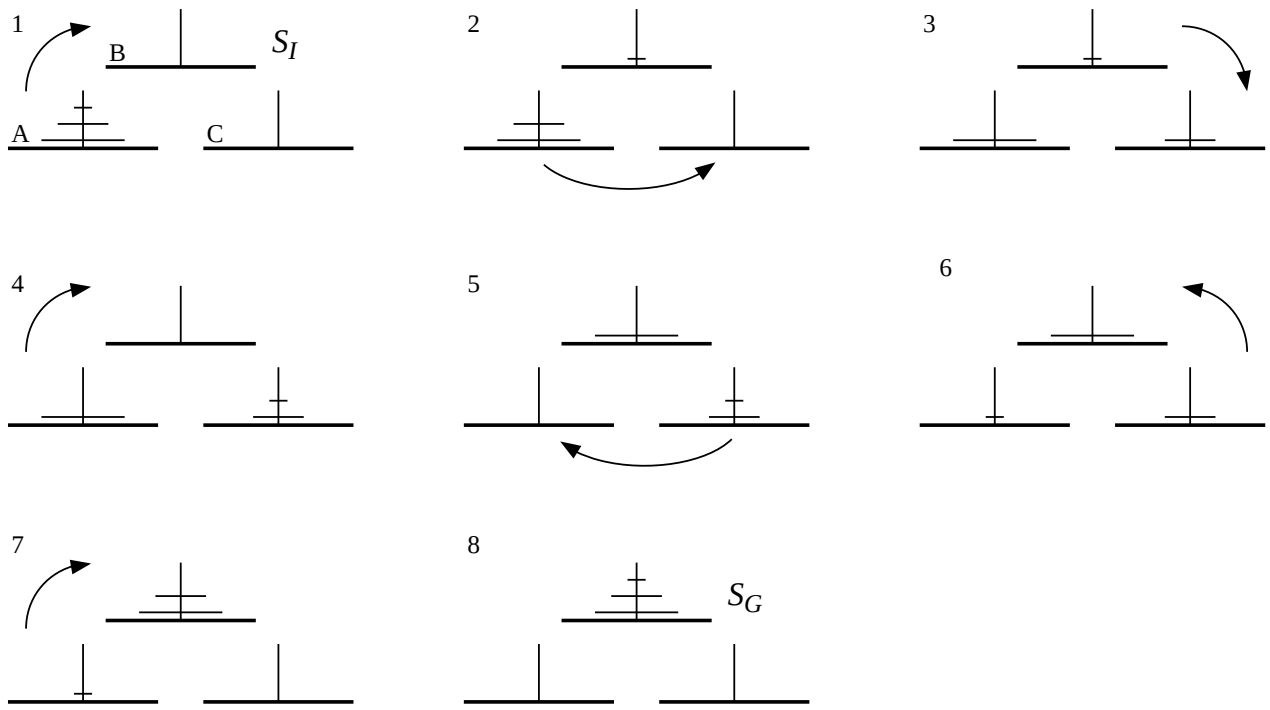

Figure 12: The iterative method for the towers-of-Hanoi task. Note how the smallest disk cycles around the pegs of the triangle A, B, C.

A.5 General-Task Decomposition

The *general-task decomposition* (GTD) approach decomposes the task explicitly into several subtasks. Using GTD, we can solve the general task of transferring any initial state S_I to any goal state S_G, by decomposing the towers-of-Hanoi task into the subtask `make_tower`(S), which transfers any state into a single-tower state (i.e., a legal state in which all disks are located on a single peg):

```
transfer(SI , SG) =
       {make_tower(SI);
        reverse(make_tower(SG))}.
```

The operator `reverse` reverses a plan by applying the inverse of all move operators in reverse order: The `make_tower(S)` task, however, is decomposed easily into n tasks that solve the classic version for m disks, $m = 0, 1, \ldots, n-1$, because transforming any state S into a single tower involves creating a tower of only the smallest disk, then transferring that disk to the top of the second smallest disk (which must be free), then transferring the tower of the two smallest disks on top of the third smallest disk, and, thus, eventually transferring a tower of $n-1$ disks onto the largest disk. Hence, we can use any of the previous methods for the classic towers-of-Hanoi problem to solve this general towers-of-Hanoi task. Note that the decomposition described here resembles the use of a macro-operator problem solver [22], whose single operator transfers any state into one intermediate state. Unlike the specific RTD solution presented in Appendix A.2, the more general GTD method works for any initial or goal configuration in the towers-of-Hanoi task.

Table 5: Time and space comparisons for different problem-solving methods for the classic towers-of-Hanoi task. (n = number of disks; k = number of pegs [$k = 3$ for the classic version])

Problem-solving method	Solution quality, number of moves	Optimal solution	Internal time	Internal space[a]	Metatime	Metaspace
chronological backtracking	$\Omega(2^n)$	no	$O(k^n)$	$O(k^n)$	up to $O(k^n)$	constant
recursive-task decomposition	$O(2^n)$	yes	$O(2^n)$	0	$O(2^n)$	$O(n)$
iterative method	$O(2^n)$	yes	$O(2^n)$	constant	constant	constant
piece-oriented concurrent method	$O(2^n)$	yes	$O(2^n)$	0	constant	constant
general-task decomposition	$O(2^n)$	no	$O(2^n)$	constant	$O(n)$	constant

[a] In units of game size.

A.6 Remarks on the Methods for the Towers of Hanoi Task

We have outlined five problem-solving methods for the towers-of-Hanoi task. Inherent in the design of knowledge-based systems from reusable methods are the knowledge and efficiency tradeoffs associated with the selection and configuration of an appropriate method. For each method for the towers-of-Hanoi task, we can ask the following questions:

1. What is the *quality* of the solution? An example of a domain-specific quality criterion is the number of moves required to execute the solution.
2. What is the *internal space* needed during execution (computation) of the solution?
3. What is the *internal time* needed to compute the solution?
4. What is the *metaspace* needed for the method agenda, assuming such an agenda is controlling the task-decomposition process and containing the task activations?
5. What is the *metatime* needed to control the method agenda (to decompose tasks, to schedule tasks, and so on)?

We summarize properties of the methods for the towers-of-Hanoi task in Tables 5 and 6.

Table 6: Comparison of different problem-solving methods for a general towers-of-Hanoi task, using robustness and additional dimensions of the solution.

Problem-solving method	Arbitrary start and goal configuration	Works for a different number of pegs	Works for a different number of disks	Takes advantage of additional pegs[a]	Remarks on method/solution
chronological backtracking	yes	yes	yes	yes	little knowledge needed; all solutions can be produced
recursive-task decomposition	no	yes	yes	no	a complete plan
iterative method	no	yes	yes	no	executable solution
piece-oriented concurrent method	no	yes	yes	no	concurrent computation
general-task decomposition	yes	yes	yes	no	a skeletal plan

[a] The problem-solving method can take advantage of additional pegs ($k > 3$) to improve the quality of the solution (in terms of the number of moves) without modification of the method.