Modeling Tasks with Mechanisms

Angel R. Puerta, Samson W. Tu, and Mark A. Musen
Medical Computer Science Group
Knowledge Systems Laboratory
Stanford University
Stanford, CA, 94305-5479
puerta@camis.stanford.edu

All of the buildings
and all of the cars
were once just a dream
in somebody's head
—Peter Gabriel, 1986

ABSTRACT

Building a problem solver and acquiring the knowledge needed to operate it are the two central goals of knowledge engineering. To achieve these goals, knowledge engineers construct models of the domain and of the task of interest. The various approaches used for modeling, however, have so far failed to define methods and techniques that can be applied across domains and tasks, and to produce models that can be reused in future applications. In this paper, we propose that both of these objectives can be achieved by the use of building blocks called mechanisms. We examine the composition of mechanisms and also show how these mechanisms can be manipulated to construct problem-solving methods. We present PROTÉGÉ-II, a knowledge-acquisition shell that uses problem-solving methods to drive the modeling of tasks, the automatic generation of knowledge-acquisition tools, and the control flow of the problem solver. The modeling of tasks, within the context of PROTÉGÉ-II, is illustrated with two examples: one from the game domain and another from the medical-therapy domain. In addition, we introduce the conceptual basis for a library of mechanisms that serves as a repository of reusable knowledge components.

1. Introduction

Knowledge acquisition is a modeling process. A modeling of tasks, of domains—even of experts and of users. Gone are the days when knowledge acquisition symbolized a transfer of knowledge from expert to machine. Researchers now concentrate on understanding the requirements of tasks, on analyzing the purpose of each input and output in the process of completing the task, and on acquiring a model of that process. Thus, knowledge acquisition is now a transfer of knowledge from environment—the environment of the process to be modeled—to machine, with the expert and the knowledge engineer being part of that environment.

The work presented here is rooted on the identification, several years ago, of domain-independent abstract models of problem solving[10], hereafter called *problem-solving methods*, or, for brevity, *methods*. These methods allow knowledge engineers to develop models of tasks by defining the role that each type of knowledge plays in the problem-solving process. Based on such methods, researchers have designed *method-oriented* architectures from which knowledge-acquisition tools are built[13]. For instance, ROGET[2] uses a version of the *heuristic-classification* method[5] to model diagnostic tasks. SALT[9] embodies the method of *propose and revise,* as applied to configuration tasks. PROTÉGÉ[11,12] was devised based on the method of *skeletal-plan refinement*[7]; it operates at the metalevel by generating knowledge-acquisition tools that, in turn, can be used to capture knowledge about a given task, such as clinical-trial management.

Although systems such as ROGET and PROTÉGÉ have been successful as prototypes, they also suffer from *brittleness,* a corollary of their adherence to a single problem-solving method. Naturally, the question arises of how to devise method-oriented architectures that support multiple problem-solving methods. In the spirit of that question, we are developing PROTÉGÉ-II[14,17], a knowledge-acquisition framework that functions at the metalevel, as does its predecessor, but that is not restricted to a sole method. The design of PROTÉGÉ-II is based on the concept of a *mechanism.* We postulate that mechanisms—procedures of a grain size finer than that of methods—can serve as building blocks to compose methods, and can be reused in the composition of different methods.

In this paper, we examine the notion of a mechanism, as it is represented in PROTÉGÉ-II, and the process of modeling a task in that system using mechanisms. In Section 2, we commence by stating what are our views of the concepts of tasks, subtasks, methods, and mechanisms; we then identify the components of mechanisms. Section 3 describes the operations allowed on mechanisms, and shows how these operations permit the composition of methods. In Section 4, we discuss approaches to indexing an on-line library of mechanisms to allow easy retrieval. In Section 5 we elaborate two examples of task modeling with mechanisms: one for a simple task to demonstrate the modeling process, and one for a complex task to illustrate the use of mechanisms in a large-scale real-world problem. Section 6 discusses accomplishments and limitations of PROTÉGÉ-II, and compares this system to other multiple-method architectures.

2. Tasks, Methods, and Mechanisms

Figure 1 presents an overview of the PROTÉGÉ-II framework. Given a task for which an advice system is to be developed, knowledge engineers use the facilities of PROTÉGÉ-II to compose a method that can solve that task. The method stipulates how the task is to be modeled (i.e., how the domain concepts are represented in terms of the method). Based on the resulting task model and on additional information gathered from the knowledge engineer, a user-interface management system generates a knowledge editor through which domain experts enter the knowledge needed by the method to solve the task is entered by domain experts[18]. The task model and the knowledge acquired in the knowledge editor are stored in a knowledge base that is accessed by an advice system. The advice system reasons about the knowledge stored in the knowledge base according to the method composed by the knowledge engineer.

Clearly, the central concepts in the design of PROTÉGÉ-II are those of *task* and *method.* Although these are terms that frequently appear in the literature, it is important to define their meaning here, for their usage varies from one context to another. For our purposes, a *task* is an activity, or an abstraction of an activity, in the real world. Examples of tasks are word processing, job scheduling, and fault diagnosis. Tasks accept some type of input and produce some type of output. When a task is applied in a particular domain, the domain determines what specific type of input can be accepted and what specific type of output can be produced. Note that, although a task imposes definite input–output requirements, it does not dictate any knowledge requirements for its completion.

Consequently, a task, by itself, does not stipulate *how* to get the desired output from the supplied input. It is only when a method is employed to solve a task that such knowledge requirements are set.

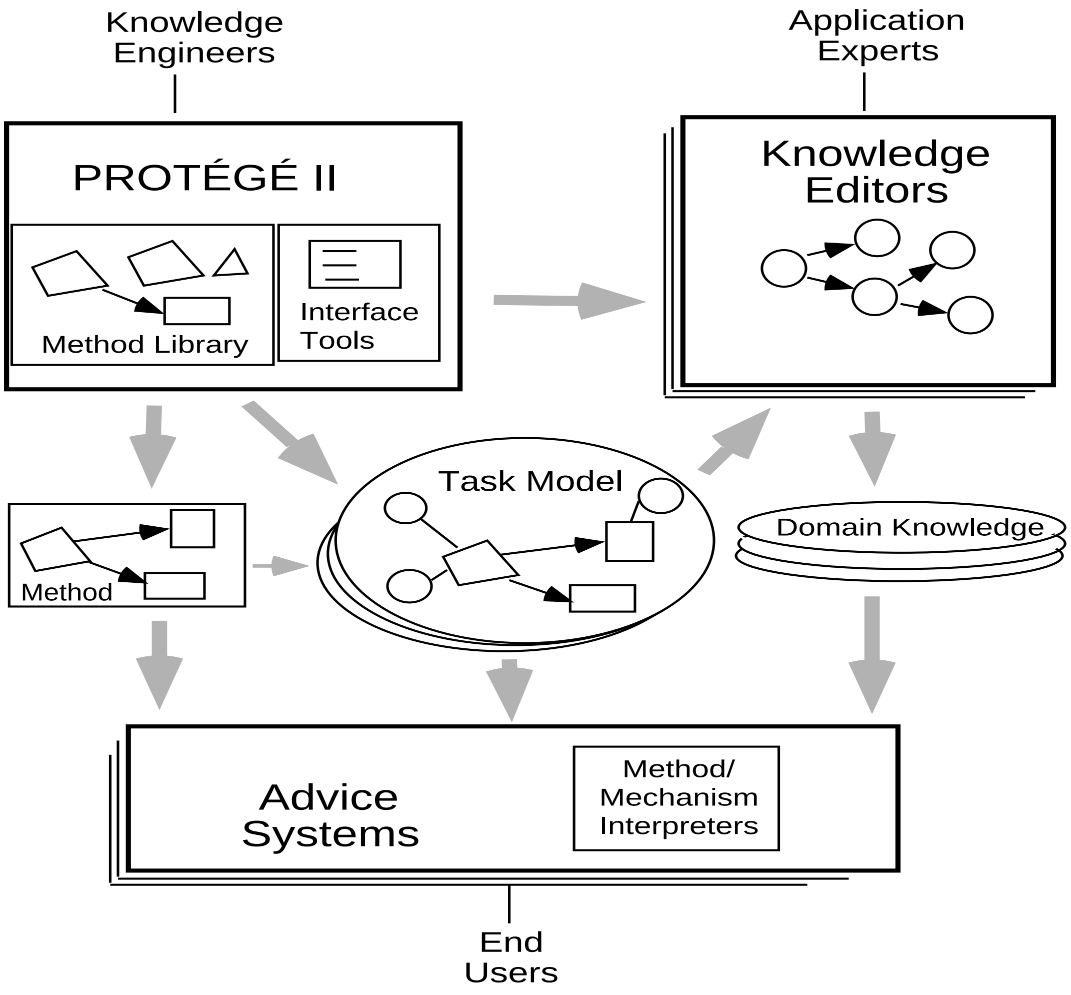

Figure 1. An overview of the PROTÉGÉ-II architecture. Knowledge engineers build task models from which knowledge editors are generated. Domain experts use these knowledge editors to edit knowledge bases that advice systems employ to offer advice to end users.

The relationship between a task and a method is illustrated in Figure 2. A *method* in PROTÉGÉ-II is a procedure that implements an abstract model of problem solving and that is applicable to a class of tasks. Applying a method to a task causes a *task decomposition* that subdivides the original task into separate subtasks. Essentially, tasks and subtasks are identical, except that subtasks occur only in decompositions. As a consequence, we can decompose a subtask further by applying a method to solve it, which produces more subtasks for which we must use additional methods. Eventually, however, the process must terminate, so that an executable solution is achieved for a task. When this point is reached, a method is then considered a mechanism. A *mechanism* is therefore a method that does not decompose a task into subtasks.

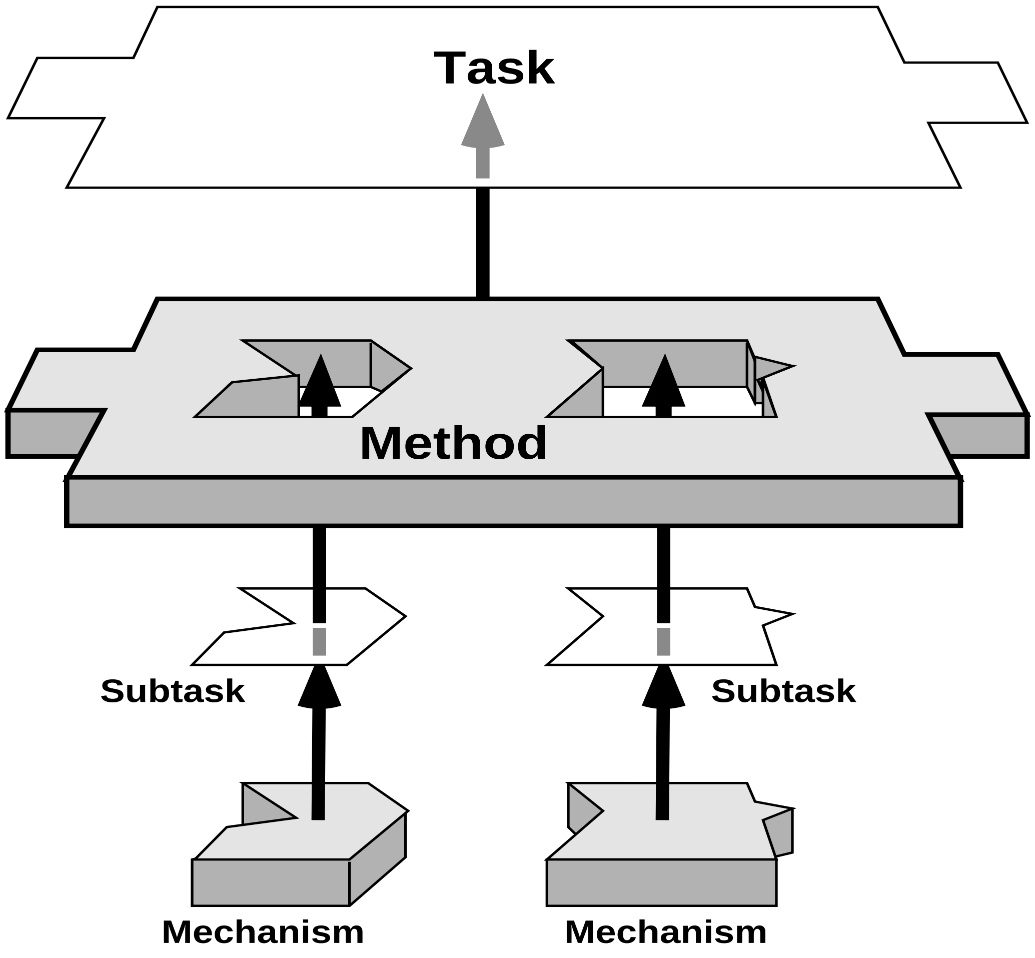

Figure 2. The relationship between tasks and methods. A method imposes a *task decomposition* on the task that it solves; the task decomposition creates a number of subtasks. These subtasks, in turn, are solved by mechanisms, or potentially by other methods that impose further task decompositions.

Since it is not possible to look into a mechanism to identify its subtasks, mechanisms are basically black boxes from the perspective of PROTÉGÉ-II. Given the inputs of a subtask, a mechanism will produce the output of that subtask. The opaqueness of mechanisms brings into question what is the appropriate mechanism granularity—the size and complexity of the mechanism that automates a particular subtask relative to other mechanisms and methods—that should be used in a task decomposition. That is, at what point should we stop the subdivision of tasks into subtasks, and use mechanisms, instead of methods, to solve such subtasks? The overriding principle in determining the granularity of a mechanism is the effect on that mechanism's reusability that a task decomposition will cause. Thus, if decomposing a method into subtasks does not provide corresponding mechanisms for each subtask that can be easily reused, the method is not decomposed and it becomes a mechanism. Also note that although we associate mechanisms with subtasks throughout this document, as it is most often done in practice, there is no restriction on applying mechanisms directly to tasks that have not been decomposed by a method. Thus, mechanisms can be viewed as building blocks that can act to solve tasks either individually, or collectively, as part of a more complex structure called a method.

The task decompositions created by methods, as illustrated in Figure 2, merely enumerate the subtasks that must be solved. There is in the decomposition itself neither an indication of the execution order of the subtasks, nor any other control specification, such as recursion or iteration, that affects the execution of the subtasks. Because mechanisms are limited to solving the resulting subtasks and must be kept method independent for reusability, the control details can be present only at the method level. Consequently, there is a clear separation of the control specification for a method and the input–output specification for a task, which is rippled down to the mechanism level through the task decomposition. As we shall discuss in Section 3, it is this partition that allows the definition of two method-manipulation operations—method configuration and method assembly—which form the basis for the reuse of knowledge components in PROTÉGÉ-II.

3. Operations on Methods

All methods and mechanisms in PROTÉGÉ-II share a common structure. It is necessary to examine that structure in depth to identify the components of a method, to understand the types of operations allowed on methods, and to show how these operations drive the problem-solving and task-modeling activities. In particular, mechanisms are made up of several distinct components, and methods are made up of a superset of the mechanisms' components. Thus, it is useful to classify these components into two categories: method-level components and mechanism-level components. In this section, we examine both component levels and describe two types of operations on methods.

3.1 Mechanism-Level Components

A mechanism has the following components: (1) a input–output declaration, (2) a data model, (3) a control-flow configuration, and (4) an interface specification (Figure 3).

The input–output declaration of a mechanism restricts that mechanism to a target class of tasks that the mechanism can solve. It does so by explicitly creating input–output requirements that must be met by any given task to belong to the mechanism's target class. One input that is common to all mechanisms is a set—possibly empty—of semantic constraints that define relationships between inputs and outputs. Whenever these constraints are violated, the resulting output for a given set of input data is not considered to be a legal solution to the task that the mechanism solves. For example, a semantic constraint could state that "output C must not be more than three times input A if input B is greater than *x*." In this manner, the behavior of the mechanisms is partially specified and the set of possible outputs is restricted.

In addition, the input and output data must be represented by a collection of data structures. The data-model component represents those data structures. This component further specifies a mechanism's target class of tasks by limiting that mechanisms's applicability to only those tasks for which the input and output data can be represented with the mechanism's data model. Furthermore, the data model is a driving factor in the process of task modeling. A knowledge engineer will be able to specialize the data structures of the data model in a mechanism to define domain concepts, but will not be able to augment, or modify, the model to define domain concepts in a format different from the one that the mechanism can process. For example, a mechanism with a temporal data model can be specialized so that all time intervals are single time points. The data model in such mechanism, however, cannot be modified such that processes are associated with more than one time interval.

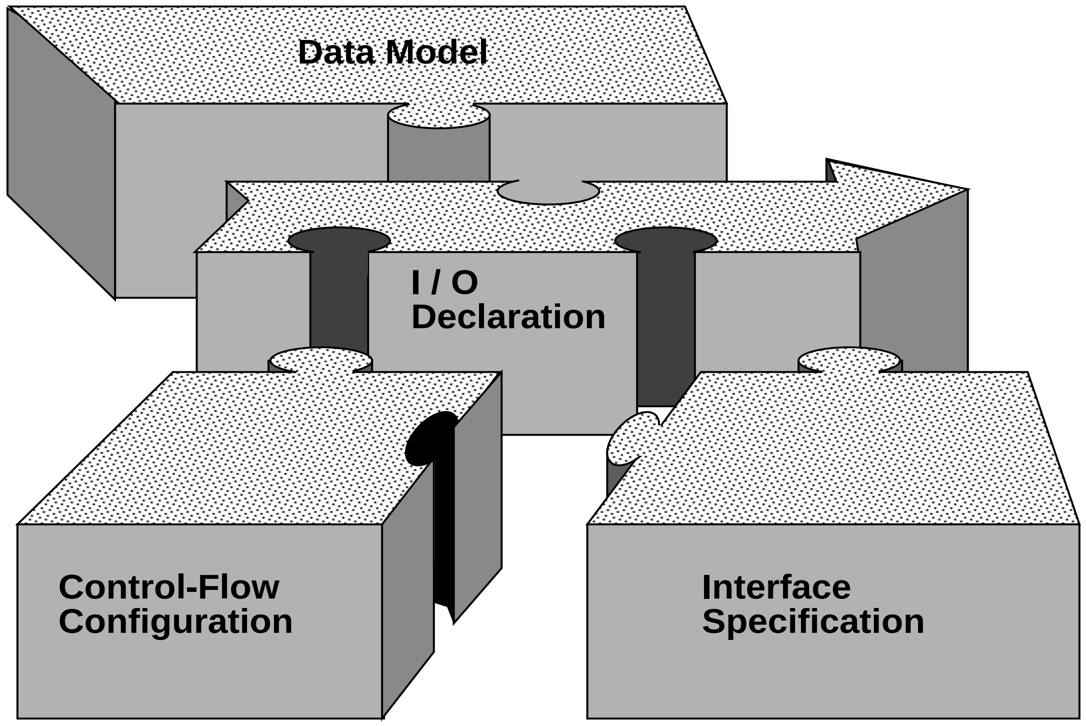

Figure 3. The components of a mechanism. A mechanism is a modular structure that allows the interchange of its components with components of other mechanisms.

The third component of a mechanism is the control-flow configuration. In the most general of terms, this component directs the transformation of the input data into output data. Given that mechanisms are stand-alone entities, there is no common control structure shared by all mechanisms. The control-flow configuration imposes knowledge requirements on the mechanism's target class of tasks. The requirements result from the mechanism's need to know under which circumstances the flow must be altered, or branched, during execution of the mechanism. Therefore, tasks for which the knowledge required by the control-flow configuration of a method is not available cannot belong to the mechanism's target class of tasks.

The final component of a mechanism is the interface specification. This component implements a user interface that allows a knowledge engineer to develop a model of a task in the terms that the corresponding mechanism dictates. Thus, the interface conveys to the user all the input–output and knowledge requirements of a mechanism. As shown in Figure 1, task models developed with this component are the basis for the generation of knowledge editors in PROTÉGÉ-II: After the task models are incorporated into a knowledge base, they are used to guide the reasoning process of the corresponding advice system.

Conceptually, we view a mechanism as a four-piece jigsaw puzzle, as illustrated in Figure 3. It is possible to substitute one piece in the puzzle for another that also fits the rest of the mechanism components. As an example, one mechanism can have more than one interface specification for task modeling, with the choice of interface left to the user. Similarly, the data model can be changed. There is, however, a fundamental difference between changing an interface specification in a mechanism and changing any other component. When one interface specification is substituted for another, only the presentation of the mechanism to the user has been modified; the mechanism's contents remain unaltered. Thus, there is still only one mechanism. In contrast, when a component other than the interface specification is substituted, the contents of the mechanism have been changed, so a different mechanism has been created.

3.2 Method-Level Components

There is only one difference between the list of components of a mechanism and that of a method. In addition to the four components already described, a method has a *task decomposition* as a separate component. The task decomposition enumerates the subtasks that must be solved when the method is executed. Each of the subtasks, as illustrated in Figure 2, is solved by a different mechanism. Whereas the task decomposition in a method is fixed, the mechanisms used to solve the subtasks are not. All mechanisms that solve a given subtask can be interchanged with one another, assuming that they use compatible data models. The interchange of mechanisms is desirable in some instances for reasons of efficiency, ease of generation of the knowledge editor, and development of the problem solver.

The task-decomposition component provides an explicit connection between the method and the mechanisms that solve the subtasks. This connection has important implications for the other method components. First, the input–output declaration of a method is, with one possible exception, a subset of the union of declarations of the lower-level mechanisms. Each input to a method translates into an input for one, or more, of the subtasks in the task decomposition, and, ultimately, into an input for the mechanisms that solve the respective subtasks. The exception occurs when an input to a method affects only the control-flow configuration at the method level. The connection between an input at the method level and an input at the mechanism level is not necessarily a simple mapping. A transformation may take place to convert the input at one level to the input at another level. Such transformations may be required by the data model that the respective methods and mechanisms assume. A similar process takes place with the outputs of a method. Second, the data model of a method restricts the types of data models that the mechanisms solving the subtasks can apply: Only those mechanisms compatible with the method's data model can be used. The mechanisms, however, may specialize the method's data model for their own purpose. Third, the method's interface specification is composed of the interface specification for each of the mechanisms applied to each subtask.

The only method component that is substantially different from its mechanism counterpart is the control-flow configuration. We noted previously that the task decomposition of a method does not specify any control of the execution of the subtasks. The control-flow configuration of a method determines all the control specifications for the execution of the subtasks as a group, such as execution order, recursion, and iteration. The control for the solution of any individual subtask, however, is left to the mechanism that solves that subtask. In PROTÉGÉ-II, we have standardized the execution of the subtasks of methods by using an agenda-based execution system that is shared by all methods[24].

3.3. Method Configuration

The most common operation that knowledge engineers perform with methods is method configuration. In this operation, a knowledge engineer starts with an existing method, which has been selected for a given task, and chooses individual mechanisms to solve each of the subtasks of the method's task decomposition. The actual work needed to configure a method, however, is more complex than simple selection of appropriate mechanisms; it encompasses verifying that the relationships among method-level and mechanism-level components are valid. From the perspective of Figure 3, it requires the knowledge engineer to make sure that all the pieces of the puzzle of mechanisms and methods fit together.

The method-configuration operation involves two components: the data model, and the interface specification. The control-flow configuration is not affected because the method- and mechanism-level control components are independent. The input–output declarations follow automatically from the task decomposition. In the case of the data model, all mechanisms used for subtasks generally assume the method's data model. Any specializations, or modifications, of the method's data model at the mechanism level, and any input and output transformations required for compatibility of data between method and mechanisms, are part of the configuration process.

Furthermore, the interface specification for task modeling must be composed from the interface specifications of the mechanisms. In the simplest of cases, the method's interface specification would be the aggregate of all specifications for the respective mechanisms. More often than not, however, the knowledge engineer must expend additional effort to incorporate the separate specifications into a single method-level interface. This procedure may include experimenting with different mechanism-level interface specifications to find the right combination for the method-level interface.

3.4 Method Assembly

A second, more complex operation performed with methods is method assembly. This operation entails the virtual creation of a new method; it demands the development of all components of a method except for the interface specification. A knowledge engineer starts this operation by constructing a task decomposition on which the rest of the method is built. From the task decomposition, the input–output declaration is derived; then, a data model is designed for the method. Finally, the control-flow configuration for the execution of the subtasks is completed. The interface specification is not completed until the method is configured.

PROTÉGÉ-II is geared much more toward method configuration than toward method assembly. The system includes a library of mechanisms and methods that should be sufficient, in many cases, to allow users to build problem solvers by performing only method configuration. This library and the problems involved in the indexing of mechanisms and methods are discussed in Section 4. We are manually performing most of the tedious work of method assembly for the library. We are identifying mechanisms and methods from expert systems previously developed in our laboratory, and are incorporating them into the framework of the library of mechanisms.

4. Storage and Retrieval of Mechanisms and Methods

Method configuration in PROTÉGÉ-II is carried out using the *library of mechanisms,* which contains not only mechanisms and methods as basic elements, but also tasks, ontologies, and task models as supporting elements for indexing, storage, and retrieval of methods and mechanisms. The purposes of the library are (1) to provide indexing facilities for mechanisms and methods that allow easy identification and retrieval of groups of candidate mechanisms for a task, and (2) to offer supporting information that guides knowledge engineers in the selection of mechanisms and methods from the retrieved candidate groups. The library is built on the principles of using multiple indexing strategies and of incorporating current instances of method configuration and task modeling into the library for future usage.

Multiple indexing for mechanisms and methods is similar to indexing for a library of books. In the latter, a search for library materials can be done by title, by author, or by topic. Any one of these three search strategies can be the most effective in producing the desired library materials, depending on the goals of the search. We envision a similar situation in the search of candidate methods by a knowledge engineer in PROTÉGÉ-II. The parallelism with a library of books, however, ends at the indexing level. Whereas the search of a library of books by one user does not typically produce any tangible benefit to other users who may need to conduct similar searches, the library of mechanisms in PROTÉGÉ-II is designed such that the experiences of one user can be saved for the benefit of future users. Thus, the library of mechanisms grows not only by the addition of new materials, but also by the use of existing ones.

Figure 4 depicts the organizational structure of the library of mechanisms. There are two main groups of materials kept in the library, each of which plays a different role in aiding a user during the method-configuration process. The first group stores mechanisms, methods, and tasks, and supports the function of extracting subsets of methods and mechanisms that fit certain user-specified criteria and that can be considered candidates to solve the task at hand. Consequently, indexing and search strategies are important in this group. The second group stores domain ontologies and task models, and supplies information that allows users to make an informed decision when choosing among the candidate methods and mechanisms produced by the search functions. These two groups are discussed in detail in Sections 4.1 and 4.2, respectively.

More than any other feature of PROTÉGÉ-II, the design of the library of mechanisms reveals our emphasis on the knowledge engineer as the principal user of the system. We exploit the skills of such users to navigate the difficult processes of constructing a problem solver from mechanisms, developing a task model, and generating a knowledge editor for the acquisition of knowledge for the problem solver.

4.1 Group 1: Mechanisms, Methods, and Tasks

The selection of an appropriate method is probably the most critical part of the construction of a knowledge editor and problem solver for a given task. Method configuration is an operation that demands from the knowledge engineer a thorough understanding of how the pieces of a method fit together and a keen judgment for whether substitution of one component for another will be beneficial. It is natural then that PROTÉGÉ-II emphasizes functionality that assists the user in making these important decisions. Group 1—tasks, mechanisms, and methods—comprises the materials from which users will choose the needed components for method configuration. The library of mechanisms implements several indexing strategies that permit the organization and retrieval of such materials such that users have a clear avenue to identify sets of candidate mechanisms.

The components of a mechanism form a basis for the indexing of methods and mechanisms. The search for candidate mechanisms can often take the form of a search for a specific type of component in a mechanism. Therefore, we have identified three main indexing strategies for mechanisms: (1) indexing by data model, (2) indexing by input–output declaration, and (3) indexing by task. The first two strategies are directly related to mechanism components; the third strategy is related to the past use of the input–output declaration component.

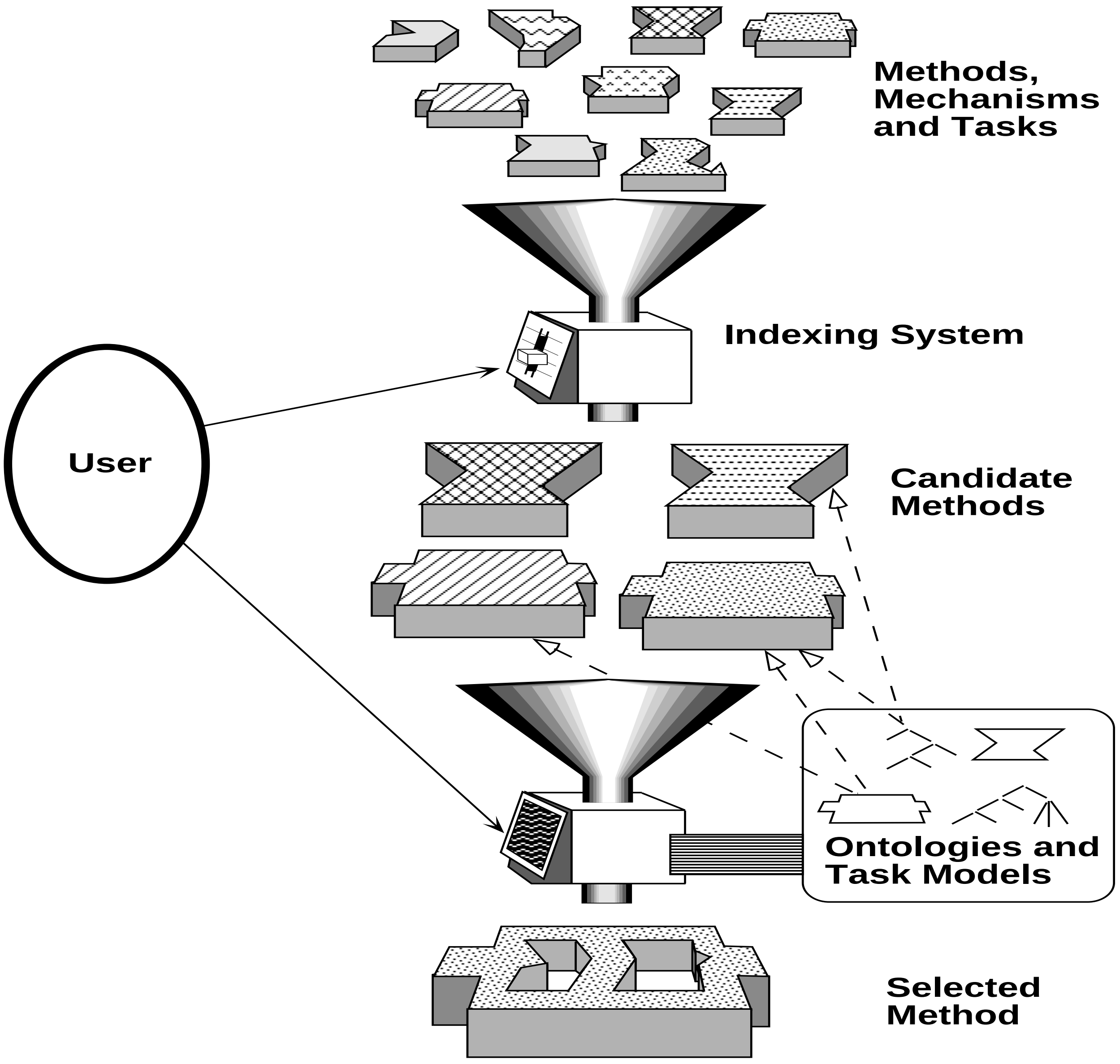

Figure 4. The library of mechanisms of PROTÉGÉ-II. An indexing system aids users in identifying a set of candidate methods to solve a given task. The user can the access information about domain ontologies and task models to make a final selection from the set of candidate mechanisms.

The indexing of methods and mechanisms by their data model is based on the principle that such models are good limiting factors of the applicability of mechanisms and methods to individual tasks. Thus, the data model is a clear delimiter of the target class of tasks that a mechanism can solve. For example, in many medical-domain tasks, the use of temporal data, such as laboratory-test results, is essential. Consequently, only mechanisms whose data model accepts temporal data can be applied to this domain[21]. In a search for candidate mechanisms for tasks of this type, the first logical step would be to limit the search to those mechanisms with the appropriate temporal data model.

In many other instances, however, there is no such clear choice of data type. Instead, what is known is the task to be solved, the available inputs, and the desired outputs. In such cases, it is more effective to match the known input–output characteristics of the tasks (e.g., temporal data) to the input–output declarations of mechanisms. Note that this search approach is likely to produce a superset of the actual set of candidate mechanisms, because some of the members of the superset may require additional inputs that are not available for the given task. To counter this problem, a complementary indexing strategy is defined: indexing by tasks. The library of mechanisms represents tasks as library elements that have a name, a textual description, a set of inputs, a set of outputs, and a set of method–task links. Searches for mechanisms and methods under this indexing strategy take the form of a lookup of task names by the knowledge engineer to find task descriptions that resemble the task to be solved. Selecting any one task allows the knowledge engineer to see what methods and mechanisms are *linked* to that particular task. A method–task link associates a specific task to a set of candidate mechanisms and methods for that task. There are three ways in which such links can be created. First, links to methods and mechanisms can be specified when the particular task is stored in the library. Second, links to tasks can be created when new mechanisms or methods are stored. Third, links to tasks can be created as a result of successful application of a method, or mechanism, to such tasks. In the first two cases, the links are considered *potential* links, because there is still no concrete evidence that the links can actually lead to a successful implementation in PROTÉGÉ-II. In the third case, the link is *definite,* because such concrete, historical evidence does exist. Indexing by tasks tends to produce subsets of the actual set of candidate mechanisms because the library may contain mechanisms and methods that can solve the given task, but that have not yet been linked to that task.

Another important element of group 1 is the *index of interface specifications.* This index is separate from the three principal ones already discussed. It is not used to define sets of candidate mechanisms. Its purpose is to organize the information that is needed for a user to select an interface specification for a mechanism. The index of interface specifications also contains links of interfaces to mechanisms. These links are all *definite* links, in the sense that the interface specifications stored in the library have been built specifically for the linked mechanism, or have been used with the linked mechanism. The concept of potential links does not apply to this index. The interface specification and the control-flow configuration are not considered good limiting factors to guide a search for candidate mechanisms; thus, there are no indexing strategies designed for these components.

4.2 Group 2: Domain Ontologies and Task Models

Despite the effectiveness of multiple indexing strategies to reduce the set of candidate mechanisms to an ideal minimum, there remain difficult decisions for a knowledge engineer who must choose single methods and mechanisms from the candidate sets. Recall from Figure 1 that the aim of the PROTÉGÉ-II user is to configure a method with which to build a task model to generate a knowledge editor. There is a clear paradox, however: to construct the task model of PROTÉGÉ-II, the user needs a single method, but to select a single method, the user requires a model of the task, at least at the cognitive level. A process of *task analysis* is carried out by the knowledge engineer before and during the library-search activities and method-configuration operations. Task analysis can take many forms. For instance, in systems such as Spark[8], task analysis is a complex, well-organized activity that must produce definite outputs before method configuration can take place. In contrast, PROTÉGÉ-II has vague delimitations where and when task analysis takes place. In some cases, task analysis can be an entirely cognitive process, where the knowledge engineer probes the environment to develop a mental model of the task. In other cases, extensive use of the library of mechanisms may be necessary, in addition to the development of the mental model. It is within the context of task analysis that the library of mechanisms includes materials such as domain ontologies and task models. These elements not only add to the retrieval capabilities of the indexing strategies, but also aid the user in selecting appropriate methods and mechanisms from the candidate sets.

Domain ontologies in the library of mechanisms provide relevant information to users about previously modeled application areas; they organize and classify domain concepts, giving the user an insight into the relevant terms in the domain of interest. For example, a domain ontology can include a classification of human diseases, or an organizational chart of a university. By examining a domain ontology, the knowledge engineer can determine the applicability of specific data models, input–output declarations, and interface specifications of candidate methods. Links between domain ontologies and mechanisms, and between domain ontologies and methods, are similar to the links that were described for tasks in the library. Domain-ontology links can be *potential* or *definite*, and links are created in the same way as with the group 1 elements. Thus, through the use of the information stored in the domain ontologies, knowledge engineers can prune the sets of candidate mechanisms—retrieved with the various search strategies—by discarding those mechanisms that do not appear appropriate for the target domain and by directing their attention to candidate mechanisms that are linked to the target domain.

Task models—the representations of domain concepts in terms of a method—are included in the library of mechanisms to maintain a historical perspective on the use of the mechanisms and methods of PROTÉGÉ-II. Every task model built from a mechanism, or from a method, is stored permanently in the library with a *definite* link to the method or mechanism from which it was developed. Knowledge engineers employ the stored task models to judge the appropriateness of a mechanism for a given task, or, after they have selected a mechanism, as a starting point for the construction of new task models.

5. Construction of Task Models

We have examined the abstract concept of a mechanism, on its representation in PROTÉGÉ-II, and on the possible manipulations of that representation. We turn now to the challenge of building a model of a task after a method and its mechanisms have been chosen. The crucial question is how to transform a domain-independent problem-solving method into a task- and domain-specific problem solver. In this section, we answer that question by means of two examples. In the first one, we show the transformation of a single mechanism into a task-specific problem-solving model for a simple task. In the second example, we illustrate the transformation of a method into a task-specific problem-solving model, including the method-configuration operation, for a complex task.

5.1 The Tower of Hanoi

The Tower of Hanoi puzzle is a known problem that exemplifies the modeling of tasks with mechanisms. A common configuration of this puzzle is shown in Figure 5. A number of disks of varying diameters are stacked by size on a peg, with the largest-diameter disk occupying the bottom position. The problem consists in transferring the disks from the original peg to another peg, with the constraints that (1) only one disk can be moved at a time, and (2) no disk can be placed on top of a smaller-diameter disk.

In the overview of PROTÉGÉ-II shown in Figure 1, there are three levels of activities for task modeling: PROTÉGÉ level, knowledge-editor level, and end-user level. The PROTÉGÉ level encompasses activities that are carried out by the knowledge engineer until a knowledge editor is generated. The knowledge-editor level comprises knowledge-acquisition activities for which the knowledge editor is a medium between a user (normally a domain expert) and a knowledge base. The end-user level includes activities related to the use of the advice system, or problem solver.

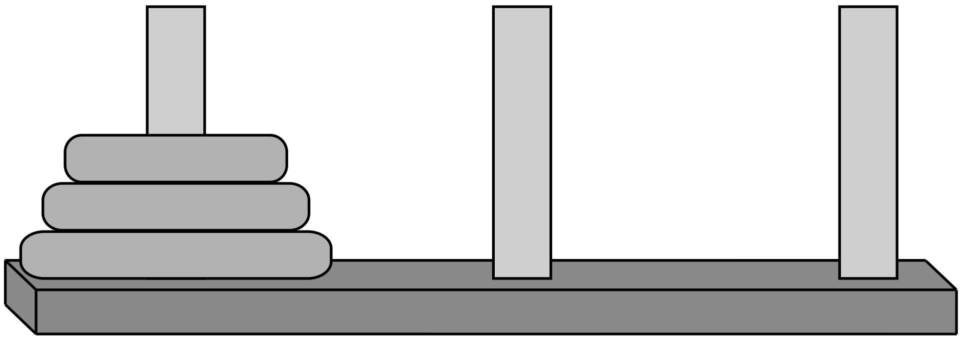

Figure 5. The Tower of Hanoi puzzle. The stack of disks must be moved from its current peg to another peg. Only one disk at a time may be moved; a larger disk may never be placed on top of a smaller one.

For the Tower of Hanoi problem, we have selected a mechanism called *chronological backtracking.* We model the task at the PROTÉGÉ level; by the time the modeling process reaches the end-user level, the mechanism will have been transformed from a general-purpose, domain-independent mechanism to a Tower of Hanoi problem solver. The chronological-backtracking mechanism is a state-transformation procedure that converts an initial state into a goal state. The input–output declaration of this mechanism contains the following inputs:

1. An initial state $\boldsymbol{S_i}$, represented by values assigned to state variables $\boldsymbol{v_1, v_2, v_3,...,v_n}$
2. A set of constraints on the type of value assignments permitted for the state variables
3. A transition function $\boldsymbol{T}(S_n)$ that takes an arbitrary state as an argument and produces a list of possible next states
4. A goal state $\boldsymbol{S_g}$, represented by values assigned to state variables $\boldsymbol{v_1, v_2, v_3,...,v_n}$
5. A (possibly empty) set of heuristics $\boldsymbol{H}(S_{nm})$ that apply to the selection of a next state from the list produced by $\boldsymbol{T}(S_n)$

The declaration also includes a single output:

1. The list of states $[\boldsymbol{S_1, S_2,...,S_m}]$ that transforms $\boldsymbol{S_i}$ into $\mathbf{S}_\mathrm{g}$

The data model of chronological backtracking allows state variables that can take on the data types most often found in programming languages, such as integer, real, string, or list. The control-flow configuration of this mechanism follows the general scheme of comparing the current state, $\boldsymbol{S_c}$, with the goal state, $\boldsymbol{S_g}$. If $\boldsymbol{S_c}$ and $\boldsymbol{S_g}$ are the same, the process halts; otherwise, the transition function is applied to generate a list of next states $[\boldsymbol{S}_{n1}, \boldsymbol{S}_{n2},...,\boldsymbol{S}_{nm}]$. A state from this list, $\boldsymbol{S}_{nk}$, is selected according to ***H****(Snm)*, and is checked against a list of states that have already been made the current state. If there is a match, $\boldsymbol{S}_{nk}$ is dropped and $\boldsymbol{S}_{nl}$, another state from the list, is checked against the list of previous current states. This verification procedure continues until a valid next state (i.e., a state that has not been a current state before) is found. At that point, the valid next state is made the current state, and the complete procedure is repeated. If no valid next state can be found, the mechanism fails. Note that, because the transition function $\boldsymbol{T}(S_n)$ is supplied by a user, chronological backtracking is a *weak* method[15].

As discussed in Section 3, it is the purpose of the interface-specification component of the chronological-backtracking mechanism to convey to the knowledge engineer the input–output and knowledge requirements of the mechanism. The interface specified by this component is used to construct the task model for the Towers of Hanoi task under the chronological backtracking method. In our example, this interface is a graphical form that allows the knowledge engineer to specify (1) a name for the state variables, (2) a data type for the state variables, and (3) a transition function. Users construct forms in PROTÉGÉ-II by using a form-specification language called FormIKA[3]. FormIKA provides grammatical structures for the declaration and layout of forms and of blanks in forms. Therefore, the interface specification of chronological backtracking is simply a set of FormIKA instructions.

Through the interface of chronological backtracking, the knowledge engineer transforms the mechanism into a general problem solver for sets of pegs and disks. The name of the state variables is given as "Peg," and the data structure for a Peg is an ordered list of integers $[\boldsymbol{D}_1, \boldsymbol{D}_2, \boldsymbol{D}_3,..., \boldsymbol{D}_n]$, where each item in the list represents a disk, and where the value of each item represents the diameter of the disk. Thus, in Figure 5, the leftmost peg is denoted as $\boldsymbol{Peg}_1$ = (3, 2, 1). The transition function, $\boldsymbol{T}(S_n)$, implements one of the requirements of the Tower of Hanoi puzzle—namely, that only one disk can be moved at a time. The list of possible next moves generated by $\boldsymbol{T}(S_n)$ is limited to the new positions that the $\boldsymbol{D}_n$ member of each peg can take.

Once the PROTÉGÉ-level requirements are specified, a user-interface management system called Mecano[18] generates a knowledge editor based on the constructed task model. The knowledge editor is an interface specification that the knowledge engineer can adapt to the characteristics of the user of the knowledge editor, the task, and the domain using the facilities of Mecano. In our example, the interface of the knowledge editor will allow the user to specify the following knowledge-editor level requirements: (1) the number of state variables (i.e., the number of pegs), (2) the initial state, (3) the goal state, and (4) the set of constraints ***C*** on the values allowed for the state variables. For the Tower of Hanoi problem, these requirements are, respectively,

1. Pegs = 3
2. $\boldsymbol{S}_i = \{\boldsymbol{Peg}_1 = (\boldsymbol{D}_1, \boldsymbol{D}_2, \boldsymbol{D}_3,..., \boldsymbol{D}_n), \boldsymbol{Peg}_2 = (), \boldsymbol{Peg}_3 = ()\}$
3. $\boldsymbol{S}_g = \{\boldsymbol{Peg}_1 = (), \boldsymbol{Peg}_2 = (), \boldsymbol{Peg}_3 = (\boldsymbol{D}_1, \boldsymbol{D}_2, \boldsymbol{D}_3,..., \boldsymbol{D}_n)\}$
4. $\boldsymbol{C} = \{D_1 > \boldsymbol{D}_2 > \boldsymbol{D}_3 > ... > \boldsymbol{D}_n \text{ for every } \boldsymbol{Peg}_n\}$

The knowledge-editor user also specifies the set of heuristics ***H****(Snm)* that allows the problem solver to make an informed selection when presented with the list of possible next states. Note that the set of constraints effectively implements the second requirement of the Tower of Hanoi problem, which is that no disk can be placed on top of a disk of smaller diameter. The chronological-backtracking mechanism has now been transformed into a Tower of Hanoi problem solver.

At the end-user level there is only one piece of information required by the problem solver: the length of the list of integers of a state variable, which translates to the number of disks in the initial state. Consequently, the work of the end user is limited to providing the integer number of disks with which to execute the Tower of Hanoi task.

Although the Tower of Hanoi puzzle is relatively simple, it illustrates clearly the process of modeling a task and building a problem solver with PROTÉGÉ-II. Starting from a domain-independent method, the problem solver is specified in increasing detail as the work proceeds from one level to the next. In Section 5.2, we shall show how the same methodology can be applied to a much more complex task in a real-world, medical problem.

5.2 Clinical-Trial Management for AIDS

Whereas the problem solver for the Tower of Hanoi puzzle illustrates the use of a single mechanism, most real-world problems require the application of methods and multiple mechanisms. In this section, we focus on one type of problem that is a main target of PROTÉGÉ-II: clinical-trial management. Clinical trials are controlled experiments where patients receive promising new treatments for a specific illness. The experimental treatments are described in a procedure called a *protocol* that physicians must follow rigorously. The number and variety of data collected in clinical trials, and the complexity of protocols, make the management of these experiments error prone and burdensome to the physicians administering the protocols. We shall describe the use of a method to build a problem solver for the management of clinical trials for AIDS patients.

As in the previous example, our discussion starts after the knowledge engineer, through a process of task analysis, determines that the method in the library of mechanisms best suited for clinical-trial management is *episodic skeletal-plan refinement* (ESPR), a version of skeletal-plan refinement[7] that supports the use of temporal data[24]. The ESPR method solves problems by starting with a general (skeletal) plan, then revising this plan repeatedly to develop more specific plans until a final, fully detailed plan is achieved. This method requires that tasks be modeled in terms of *planning entities*, *input data*, and *revision actions*[11]. Planning entities are problem-solution components that can be decomposed into other planning entities, thus forming a hierarchy. Input data are gathered from the environment affected by the problem. Revision actions are procedures that can modify instances of active planning entities. For example, in our domain of interest, the knowledge engineer can define protocols as planning entities. A laboratory-test result will be part of the input data, whereas drug attenuation can be a revision action that affects a protocol. In addition, planning entities and revision actions can have *attributes*. The role of an attribute is to attach a value to a particular characteristic of a planning entity, or action, needed for problem solving. For example, in episodic skeletal-plan refinement, all planning entities have an attribute called *duration* that determines when each entity is active in the plan. This attribute gives planing entities their temporal nature.

Figure 6 shows the task decomposition for ESPR and the mechanisms selected for each subtask. The input–output declaration of this method defines the following inputs:

1. A hierarchy of planning entities
2. A set of plan data
3. The current time

The declaration includes a single output

1. A fully detailed solution plan for the current time

As in the Tower of Hanoi example, the inputs are supplied at all three levels: the PROTÉGÉ level, the knowledge-editor level, and the end-user level. The type of inputs supplied determines how the domain-independent ESPR method can be transformed into a problem solver for management of AIDS clinical trials.

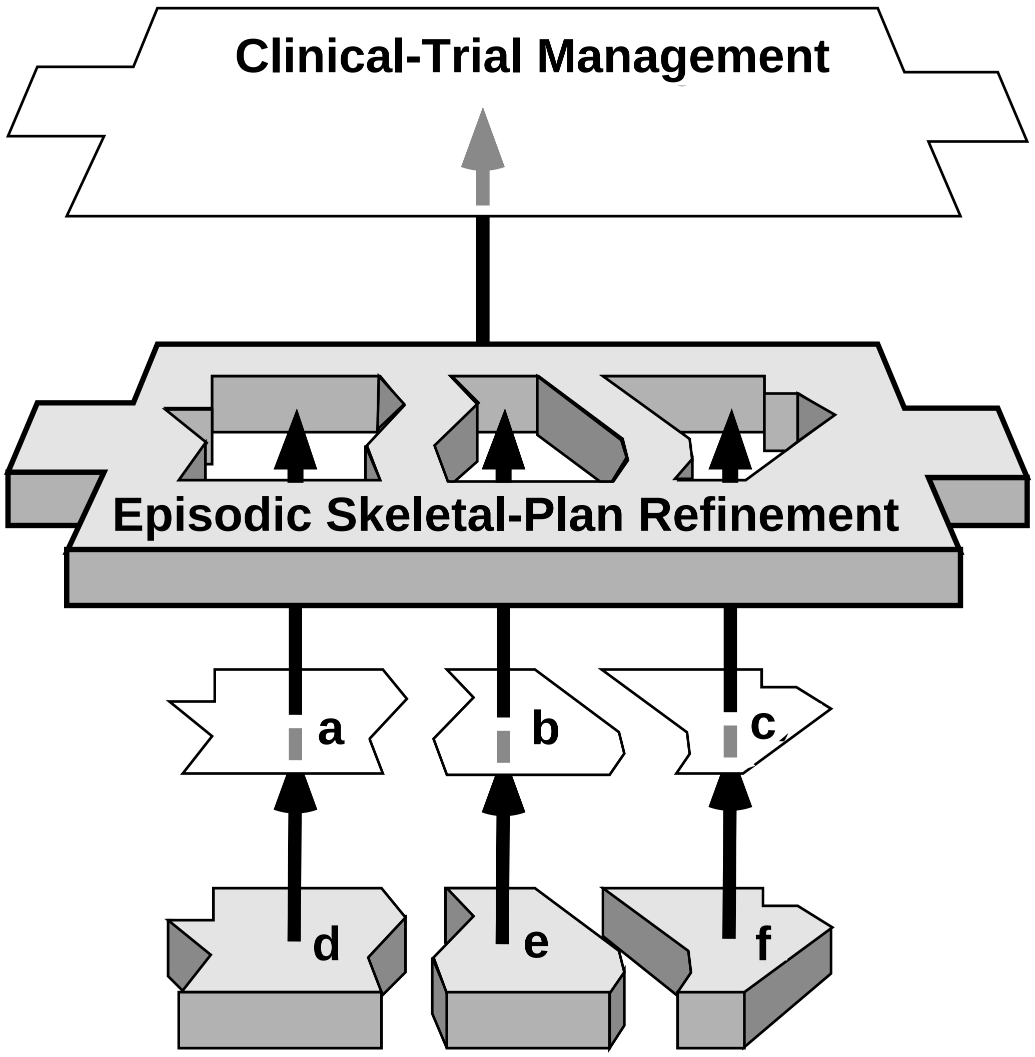

Figure 6. The task decomposition of episodic skeletal-plan refinement for the clinical-trial management of AIDS. The method identifies three subtasks: (a) plan decomposition, (b) plan generalization, and (c) plan revision. These subtasks are solved by the following mechanisms in the PROTÉGÉ-II library: (d) instantiate and decompose, (e) generalize from situation, and (f) situation-based repair.

At the PROTÉGÉ level, the knowledge engineer models the clinical-trial management task in terms of the ESPR method. The resulting task model identifies the planning entities and their attributes, declares the input data, and partially specifies the actions that affect the planning entities. There are three planning entities in our example: protocols, regimens, and drugs. They form a hierarchy that states that protocols consist of regimens—groups of drugs administered to patients in a particular order and in specific dosages—and that regimens consist of drugs. Examples of relevant attributes are name, duration, and dosage, in the case of drugs, for instance. The input data are the *case data*—a collection of data pertaining to a single patient—that include items such as blood-cell counts and body weight. Note that input data must be defined as a specialization of the temporal data model of ESPR.

The interface specification of ESPR provides an array of forms through which the user constructs the task model of the clinical-trial management task in terms of planning entities, actions, and input data. Also at the PROTÉGÉ level, the knowledge engineer employs Mecano, the PROTÉGÉ-II user-interface management system to generate a knowledge editor for use by the domain expert[18]. In this example, the generated knowledge editor consists of a graphical editor and a separate array of forms. The graphical editor lets the domain expert draw AIDS protocols as *flowcharts*, a common representation of these treatments plans in the medical domain. The forms allow physicians to stipulate values for the defined planning-entity attributes (e.g., doses of drugs), and to define how revision actions may affect such planning entities (e.g., drug-dosage attenuation in the setting of certain side effects). In this manner, at the knowledge-editor level, the transformation of ESPR into a problem solver for the management of AIDS clinical trials is completed. The remaining inputs to ESPR—the case data, and the current time—are supplied at the end-user level. End users, for this example, are the physicians who actually administer to patients the protocols stored in the knowledge base.

Note that, despite the marked difference in complexity between the Tower of Hanoi and the clinical-trial management examples, the process of building a problem solver is the same in each instance. The selection of a method, or of a mechanism, drives the modeling of the task at the PROTÉGÉ level; the inputs of the method can be supplied at any of the three levels, by knowledge engineers, by domain experts, or by end users. The transformation of a domain-independent method into a task- and domain-specific problem solver is completed at the knowledge-editor level. In sum, PROTÉGÉ-II provides a common framework for the building of advice systems useful for a wide range of problems. In turn, the use of PROTÉGÉ-II to build such advice systems results in the expansion of the library information that can be reused by future users to minimize the work required to develop new problem solvers.

6. Discussion

In this section, we compare the methodology underlying PROTÉGÉ-II to those of other approaches that seek to automate the endeavour of developing computing systems. First, we make a clear distinction between the goals of method-oriented architectures and those of automatic-programming systems. Second, we compare PROTÉGÉ-II to other method-oriented architectures and find that, despite the varying implementations, there exists a high degree of commonality among all of these systems.

6.1 Automatic Programming and PROTÉGÉ-II

The concept of putting together a problem solver in PROTÉGÉ-II by combining building blocks evokes the goals of automatic programming. The latter term can be defined broadly as the process of automating some part of the programming process[1]. Although automatic programming has been relatively successful in building small-scale programs, it has faced great difficulties achieving the same success for real-world problems. A majority of approaches to automatic programming include the use of knowledge-based tools to guide the generation of target programs. For instance, the Programmer's Apprentice project[19] seeks to automate program development by using a library of *clichés* that perform simple functions, such as comparing two numbers, or more intricate operations, such as creating reports from data sets. The Programmer's Apprentice—and knowledge-based systems for automatic programming in general— encompass knowledge about programming techniques (e.g., with clichés) and about the human activity of programming to be able to assist the programmer in that activity. These systems normally also incorporate knowledge about domains so that, in the case of the Programmer's Apprentice, the applicability of a cliché to a given task can be assessed more precisely. For a programmer, the process of creating a program with the Programmer's Apprentice consists of developing a solution method—that is, the appropriate control and data flow—using the clichés available in the system's library.

Although PROTÉGÉ-II can be thought of as automating parts of the programming process of building advice systems, its approach to automation is distinct to that of traditional automatic programming. The knowledge in PROTÉGÉ-II, in the form of problem-solving methods, concentrates on how to solve tasks, how to model these tasks, and how to acquire the necessary domain knowledge to solve a given task. There is no emphasis on understanding the activity of programming or the process of generating code, unlike the case of automatic programming. The result is that, whereas developing the solution to a task with an automatic programming system is mostly a *control-design* activity, achieving a solution with PROTÉGÉ-II is mostly a *modeling* activity. A programmer, in the Programmer's Apprentice framework, must specify *how* a task is solved—in essence an operation of method assembly. A knowledge engineer, in the PROTÉGÉ-II framework, models the solution in terms of configurable problem-solving methods that dictate how the task is solved.

In general terms, we view automatic programming as automating the method-assembly operation, unlike Protege-II, which automates the method-configuration operation. From our experience, method assembly is a much more complex operation than is method configuration. Thus, we should expect that progress in method-assembly automation will be slow compared to progress in method-configuration automation.

6.2 PROTÉGÉ-II and Other Multiple-Method Architectures

It is useful to compare the PROTÉGÉ-II system to other multiple-method architectures to find commonalities in, and to examine differences among, the various approaches. There are several architectures that, like Protege-II, have evolved from the conceptual foundations of *generic tasks*[6], *role-limiting methods*[10], and *interpretation models*[4]. Many of these architectures also share, along with PROTÉGÉ-II, Steel's view of a componential framework in which the construction of problem solvers can be analyzed in terms of *tasks, domain models and problem-solving-method*[23]. From these architectures, we have chosen three systems for comparison: (1) the Spark environment[8], (2) the EMA architecture[22], and (3) the KADS methodology[25]. Surprisingly, there is no marked difference among the four approaches in the methodology followed to develop applications starting from an arbitrary method of problem solving. Instead, differences tend to reflect the individual goals that each approach has when creating the target applications. With slight variations, all four approaches walk through the following steps, which must be iterated before an acceptable application is obtained:

1. Task analysis
2. Method configuration

3. Task modeling
4. Application generation

The first system used for comparison purposes is being developed by McDermott's group at Digital Equipment Corporation. Its designers share our view that problem-solving methods can be composed from more primitive building blocks, called mechanisms[8], and that these mechanisms can define distinct roles for the knowledge that knowledge-acquisition tools can supply. The group at Digital is developing a metatool called Spark, which will select mechanisms from a library and will combine them to construct problem-solving methods that accommodate the requirements of particular application tasks. The output of Spark is a method-oriented knowledge-acquisition tool (called Burn) that will allow users to enter knowledge in the spirit of tools such as MOLE and SALT. Although the output produced by Spark—like the output of PROTÉGÉ-II—is a knowledge-acquisition tool, Spark generates knowledge-acquisition tools with which users enter knowledge in terms of the problem-solving method assembled from mechanisms at the metalevel. PROTÉGÉ-II, on the other hand, first requires the knowledge engineer to instantiate portions of the assembled problem-solving method with domain knowledge. Consequently, the output of PROTÉGÉ-II is a task-oriented knowledge-acquisition tool with which domain specialists can enter knowledge in terms of the application area, rather than in terms of the knowledge roles provided by the method[14].

The comparison of Spark with PROTÉGÉ-II probably typifies how the goals of each system affect each design’s emphasis on one of the steps. Whereas PROTÉGÉ-II relies heavily on the skills of the knowledge engineer, Spark is tailored to a nonprogrammer. Consequently, Spark puts great emphasis on task analysis, moving the user through a formal process of analysis that produces specific outputs (an activity model and a workflow manager). Spark then maps activities from the activity model to individual mechanisms, mostly automatically. PROTÉGÉ-II, although not precluding formal processes of task analysis, lets knowledge engineers determine on their own the mapping between activities (tasks and subtasks) and mechanisms. As a result, PROTÉGÉ-II emphasizes the library of mechanisms and the problem of indexing methods and mechanisms for facile retrieval. Because, unlike the nonprogrammer who uses Spark, the PROTÉGÉ-II user is able to map tasks to mechanisms dynamically, our system includes tools that assist in the mapping process. Spark, on the other hand, includes tools that formalize the task analysis process—an approach that is possible, but not mandatory, in PROTÉGÉ-II. The mechanism-configuration problem addressed by PROTÉGÉ-II is left as an exercise for the developers of Spark.

The second system, EMA, also parallels PROTÉGÉ-II by supporting the development of knowledge-based applications based on models of problem-solving and by identifying building blocks with which the problem-solving methods can be constructed. The EMA architecture distinguishes two types of generic techniques (called G-TECs for short)—the equivalent of mechanisms. *Conceptual* G-TECs are task specific (e.g., evaluation) and are mapped to one or more problems identified in a classification of solutions. Thus, given a task, EMA searches through its classification of problems, matches the given task with a stored problem, and maps the appropriate conceptual G-TEC to the given task. The other type of G-TECs comprises *technical* generic techniques (e.g., sorting), which are task neutral and potentially can be applied to any given task. Because the mapping of tasks to conceptual G-TECs is preset, EMA does not require any explicit task analysis on the part of the user. In PROTÉGÉ-II, all mechanisms are task neutral; therefore, task analysis is essential. EMA so far has been applied to only banking tasks with which the developers are familiar. It is unknown whether EMA can function outside the class of tasks to which it has been applied already. If a new problem, outside the scope of the existing mappings of conceptual G-TECs to problems, is to be solved, the developers of EMA, and not the system's users—unlike in the other systems being compared here—would be responsible for identifying the proper mapping for the new problem.

The third reference for comparison, KADS, is a methodology for the development of knowledge-based systems[25]. The methodology covers all aspects of the development of knowledge-based systems from interviews with domain experts to the production of the executable code. It categorizes the development process by identifying three *models* (task, conceptual, and design) that must be constructed at different points of this process and four *layers* of knowledge (domain, inference, task, and strategy) that classify the type of knowledge that is manipulated in each of the models.

The KADS methodology for knowledge engineering cannot be compared directly with PROTÉGÉ-II, because the latter is a computer system. However, the methodology that PROTÉGÉ-II follows to construct applications is similar to that of KADS. As does Spark, KADS formalizes the output of task analysis by requiring the production of a *conceptual model*. It arrives at that model through a task-classification step, in the same spirit as EMA, that produces a *task model*. From the conceptual model, a *design model* is completed that eventually yields the final application. PROTÉGÉ-II does not use a task-classification scheme, although we are considering implementing one as part of the retrieval operations of the library of mechanisms. Our decision will be based on our experiences observing users who conduct searches with the library, but we do not expect such a classification to be the central point of reference for the mapping of tasks to mechanisms.

The major difference, however, between KADS and PROTÉGÉ-II is in the treatment of *tasks* within the respective methodologies. In KADS, tasks imply a task decomposition that drives problem solving. On the other hand, tasks in PROTÉGÉ-II are input–output pairs and do not prescribe a task decomposition. Such decomposition is not realized until a problem-solving method is applied to the solution of the task. Consequently, there is an inability in KADS to dynamically change the decomposition of a task into subtasks—one of the main functions present in PROTÉGÉ-II.

The existence of ample common ground shared by all approaches is encouraging. Among the key challenges that lie ahead for PROTÉGÉ-II are studies of the effectiveness and validity of mechanisms as building blocks in combining with one another and forming more elaborate problem-solving methods, the identification of appropriate storage and retrieval techniques for those building blocks, and the facilitation of the task-analysis process that plays such an important role in each of the methodologies.

7. Conclusions

Many researchers are working to develop architectures for intelligent systems that support multiple methods of problem solving. Earlier work has identified limitations of architectures based on single models of problem solving that arose from the definition of role-limiting methods[9]. Although these method-oriented architectures resulted in the implementation of several knowledge-acquisition tools, each one's usefulness was restricted to a narrow class of tasks that its particular method could solve. In addition, scientists working on method-oriented systems such as PROTÉGÉ-I[11] encountered the barriers that the method-based architecture was not sufficiently flexible to permit the stipulation of additional control knowledge at the knowledge level[16], and that users often had to resort to defining such knowledge at the symbol level—for instance, with rules[14].

The aim of PROTÉGÉ-II is to provide a common framework for the development of advice systems that overcomes the limitations of single-method architectures. Central to the development of PROTÉGÉ-II is the notion of a *mechanism*. The modular nature of the components of a mechanism lets us emphasize two key aspects of the PROTÉGÉ-II conceptual model: (1) the definition of a *building block* for the construction of problem-solving methods, and (2) the reusability of these building blocks in the construction process. The components permit a clear assembly of the building blocks into complete methods. In particular, the control-flow configuration component, which represents the run-time implementation of the method, allows us to modularly manipulate *knowledge-level control knowledge*[20]. In addition, the interface-specification component allows the separation of the execution portion of the method from the discourse portion. This separation is crucial in a system such as PROTÉGÉ-II that needs to generate knowledge editors, which are essentially interfaces between users and knowledge bases.

The emphasis on reusability translates into the design of a library to store and access the mechanisms. The library not only offers storage and retrieval services, but also aids users in identifying the appropriateness of each method and mechanism for the given task. In addition, the library stores previously developed task models that can be modified by future users to be employed in similar tasks. The library elements are indexed by several indexing strategies to facilitate library searches. The effectiveness of these strategies, however, has not been determined. The strategies have been designed based on assumptions about what factors may be relevant in the search for candidate mechanisms. It is our goal to evaluate the usefulness of each indexing strategy, and to refine, or add, strategies as we gain experience in using PROTÉGÉ-II to build systems.

An important obstacle in the development of a fully functional PROTÉGÉ-II—that is, a PROTÉGÉ-II with a well-stocked library that can be used for a significant number of real-world problems—is the identification of mechanisms and the assembly of methods. Currently, we are concentrating on that problem, and have identified ten mechanisms from expert systems constructed in the past in our laboratory. Through the assembly of these mechanisms into methods, we will gain understanding about the conditions necessary for the assembly of two or more mechanisms and for the reusability of each mechanism.

We envision a working environment for knowledge engineers, domain experts, and end users where the construction of problem solvers is a collaborative endeavour in which PROTÉGÉ-II plays the twin roles of serving as a repository for reusable knowledge components, and of providing a set of tools to access those components, to model tasks, to generate knowledge-acquisition tools, and to edit knowledge bases for a wide spectrum of domains. A system with the capabilities of PROTÉGÉ-II, along with others that share its philosophy, has the potential to greatly enhance the productivity of knowledge engineers, and to improve significantly the arduous process of expert-system development.

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, and by a gift from Digital Equipment Corporation.

We thank John Egar for the artwork, and Lyn Dupré for her editing of a previous draft of this paper. We have benefited enormously from discussions with John Dawes, John Egar, Henrik Eriksson, Amir Najmi, Gretchen Purcell, Yuval Shahar, Eckart Walther, and James Winkles.

References

1. D.R Barstow, "An experiment in knowledge-based automatic programming," *Artificial Intelligence*, **12**, 73–119 (1979).
2. J. S. Bennett, "ROGET: A knowledge-based system for acquiring the conceptual structure of a diagnostic expert system," *Journal of Automated Reasoning*, **1**, 49–74 (1985).
3. A. Bennett, *A Form-Based User Interface Management System for Knowledge Acquisition*, Master's Thesis, KSL-Report 90-43, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 1990.
4. J.A. Breuker and B.J. Wielinga, "Use of models in the interpretation of verbal data," in *Knowledge Acquisition for Expert Systems: A Practical Handbook*, A.L. Kidd, editor, Plenum, London, 1987, pp. 17-44.
5. W.J. Clancey, "Heuristic classification," *Artificial Intelligence*, **27**, 289–350 (1985).
6. B. Chandrasekaran, "Generic tasks for knowledge-based reasoning: high-level building blocks for expert system design," *IEEE Expert*, **1**(3), 23–30 (1986).
7. P.E. Friedland and Y. Iwasaki, "The concept and implementation of skeletal plans," *Journal of Automated Reasoning*, **1**, 161–208 (1985).
8. G. Klinker, C. Bhola, G. Dallemagne, D. Marques and J. McDermott, "Usable and reusable programming constructs," K*nowledge Acquisition*, **3**, 117–135 (1991).
9. S. Marcus and J. McDermott, "SALT: A knowledge acquisition tool for propose-and-revise systems," *Artificial Intelligence*, **39**, 1–37 (1989)
10. J. McDermott, "Preliminary steps toward a taxonomy of problem-solving methods," In *Automating Knowledge Acquisition for Expert Systems*, S. Marcus, editor, Kluwer Academic, Boston, 1988, pp. 225–256.
11. M.A. Musen, *Automated Generation of Model-Based Knowledge-Acquisition Tools*, Pitman, London, 1989.
12. M.A. Musen, "An editor for the conceptual models of interactive knowledge-acquisition tools," *International Journal of Man–Machine Studies*, **31**, 673–698 (1989).
13. M.A. Musen, "Conceptual models of interactive knowledge-acquisition tools," *Knowledge Acquisition*, **1**, 73–88 (1989).
14. M.A. Musen and S.W. Tu, *A Model of Skeletal-Plan Refinement to Generate Task-Specific Knowledge Acquisition Tools*, KSL-Report 91–05, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 1991.
15. A. Newell, "Heuristic Programming: Ill-structured problems," In *Progress in Operations Research, III*, J. Aronofsky, editor, Wiley, New York, 1969.
16. A. Newell, "The knowledge level" *Artificial Intelligence*, **18**, 87–127 (1982).

17. A.R. Puerta, J.W. Egar, S.W. Tu and M.A. Musen, “A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools,” *Knowledge Acquisition*, **4**, 171–196 (1992).

18. A.R. Puerta, J.W. Egar and M.A. Musen, *Automated Generation of Adaptable Knowledge-Acquisition Tools with Mecano*, KSL-Report 91–62, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 1991.

19. C. Rich and R.C. Waters, “The programmer’s apprentice: a research overview,” *IEEE Computer,* **21**(11), 82–89 (1988).

20. G. Schreiber, H. Akkermans and B.J. Wielinga, “On problems with the knowledge level perspective. In *Proceedings of the Fifth Knowledge-Acquisition for Knowledge-Based Systems Workshop,* J.H. Boose and B.R. Gaines, editors, Banff, Alberta, Canada, 1990, pp. 30.1–30.14.

21. Y. Shahar, S.W. Tu and M.A. Musen, “Knowledge acquisition for temporal-abstraction mechanisms,” *Knowledge Acquisition.*, **4**, 217–236 (1992)

22. S. Spirgi, D. Wenger and A.R. Probst, “Generic techniques in EMA: A model-based approach for knowledge acquisition,” In *Proceedings of the Sixth Knowledge-Acquisition for Knowledge-Based Systems Workshop,* J.H. Boose and B.R. Gaines, editors, Banff, Alberta, Canada, 1991, pp. 31.1–31.13.

23. L. Steels, “Components of expertise,” *AI Magazine,* **11**(2), 28-49 (1990).

24. S.W. Tu, Y. Shahar, J. Dawes, J. Winkles, A.R. Puerta and M.A. Musen, “A problem-solving model for episodic skeletal-plan refinement,” *Knowledge Acquisition,* **4**, 197–216 (1992).

25. B.J. Wielinga, A. Th. Schreiber and J.A. Breuker, “KADS: A modelling approach to knowledge engineering,” *Knowledge Acquisition*, **4**, 5-54 (1992).