A Multiple-Method Knowledge-Acquisition Shell for the Automatic Generation of Knowledge-Acquisition Tools

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

Abstract

The use of predefined models of problem-solving methods is receiving considerable attention from researchers in the area of knowledge acquisition. Using these models, developers of knowledge-acquisition tools are able to prescribe the roles in which knowledge is used in completing a given task. A number of *method-oriented* architectures based on a single problem-solving method have been developed by various research groups. Because the methods are domain-independent, method-oriented architectures are limited by the fact that knowledge roles that depend on domain-specific considerations cannot be represented using the model of problem solving. In addition, the interface between the knowledge-acquisition tool and the application expert cannot adequately convey the role of each knowledge type in the task model. PROTÉGÉ-II is a knowledge-acquisition shell that we are building to generate knowledge-acquisition tools automatically without presupposing a specific model of problem-solving. The shell manages a library of mechanisms—procedures of grain size smaller than that of problem-solving methods. Mechanisms can be combined in PROTÉGÉ-II to construct problem-solving methods and to define the roles of knowledge that depend on domain considerations. Furthermore, PROTÉGÉ-II utilizes the concept of *adaptation* in interfaces to allow the knowledge engineer to produce interfaces that are task- and domain-specific. In this paper, we present the PROTÉGÉ-II shell and examine the components of its architecture. We also demonstrate the use of PROTÉGÉ-II with a running example, and discuss the design techniques used to overcome the limitations of method-specific architectures.

1. Method-Oriented Architectures

A recent focus of research in knowledge acquisition is the use of models of domain-independent problem-solving methods to construct knowledge-acquisition tools [McDermott, 1988]. These models of problem solving, such as heuristic-classification [Clancey, 1985] and skeletal-plan refinement [Friedland and Iwasaki, 1985], allow a knowledge engineer to develop a model of the task area at hand in terms of the abstract problem-solving method. Thus, task- and domain-specific knowledge-acquisition tools can be obtained from a task- and domain-independent model. Since the problem-solving models are independent of any knowledge-representation formalism, the modeling of tasks occurs at the *knowledge level* [Newell, 1982], where only the role of each type of knowledge is specified, as opposed to at the symbol level, where the representation of each type of knowledge must be described. There are several examples of knowledge-acquisition tools built from models of problem solving. ROGET [Bennett, 1985] used a specialized form of the heuristic-classification method to acquire knowledge for diagnostic tasks. SALT [Marcus and McDermott, 1989] used a *propose-and-revise* method for configuration tasks.

There is also another category comprising tools that operate at a metalevel. These tools are able to generate knowledge-acquisition tools automatically from a model of a task. Examples of this type of tool are PROTÉGÉ [Musen, 1989a; Musen, 1989b], which is method-oriented, and DOTS [Eriksson, 1990], which does not follow a given problem-solving method.

PROTÉGÉ, which was developed in our laboratory, views the problem of automating the construction of knowledge-acquisition tools as one of generating an interface to a knowledge editor that is utilized by an application

expert. The components of the interface are determined by a task model developed from the method of skeletal-plan refinement [Musen and Tu, 1991]. PROTÉGÉ makes extensive use of graphical interaction modalities, such as graphical editors, which are especially well suited for capturing procedural knowledge (e.g., a flowchart diagram). Because the interfaces are based on the task model for the domain of interest, PROTÉGÉ-generated tools can guide the user through the acquisition sessions, ensuring that the knowledge captured is complete and consistent with respect to that model [Musen et al., 1987].

Like other method-oriented knowledge-acquisition tools, PROTÉGÉ suffers because its domain-independent problem-solving method cannot define the role of domain-dependent control knowledge. For example, in the domain of cancer therapy, there is no way for the knowledge engineer to use the skeletal-refinement method to declare how the effects of various specifications for altering the dose of a drug should be combined, because the concept of a *dose adjustment* is a domain-specific one. In these instances, a knowledge engineer must fall from the knowledge level to the symbol level to enter the required knowledge in a particular representation (e.g., by specifying a production rule or by changing the ordering of rules).

Our current research aim is to identify building blocks, called *mechanisms*, of a grain size finer than that of problem-solving methods, that can be (1) combined to construct problem-solving methods, and (2) applied to define the role of domain-dependent knowledge [Tu et al., 1991]. Our approach is essentially empirical in that we hope to identify several of these mechanisms from the problem-solving methods being applied in our current development of various medical expert systems. After the mechanisms are identified, we will test their applicability to other tasks and domains, and their combinability into problem-solving methods different from the ones from which they were extracted.

Other groups are carrying out parallel research to find ways to define the roles of domain-dependent and domain-independent knowledge at the knowledge level. Examples are Chandrasekaran's research on generic tasks [Chandrasekaran, 1986], Steels' work on the componential framework [Steels, 1990], and McDermott's study of role-limiting methods [McDermott, 1988].

1.1. A New PROTÉGÉ

The use in our group of mechanisms for the generation of knowledge-acquisition tools has dictated the need for reimplementation of PROTÉGÉ. To address this need, we are developing PROTÉGÉ-II, a knowledge-acquisition shell that provides a task-modeling environment for knowledge engineers and a knowledge-editing environment for application experts. The shell is independent of methods, tasks, and domains. PROTÉGÉ-II will allow the definition of knowledge roles at the knowledge level for both domain-dependent and domain-independent knowledge. This shell will permit the construction of problem-solving methods using mechanisms as building blocks, the modeling of application tasks in terms of the constructed methods, the generation of knowledge editors based on those task models, and the acquisition of knowledge from such knowledge editors.

In creating PROTÉGÉ-II, we have to remove two fundamental types of limitations in PROTÉGÉ: limitations produced by presupposition of a problem-solving method, and limitations produced by presupposition of an interface style. We have mentioned the limitations caused by the single-method problem. The limitations imposed by the interface style, however, may be just as crucial. PROTÉGÉ has no flexibility in presenting the contents of a knowledge base to the application expert. The presentation style is fixed, and is based on the model of skeletal plan-refinement, which is domain-independent. Therefore, regardless of the task or domain, similar types of knowledge are always presented in the same style to the application expert. The lack of flexibility in the style of the interface in PROTÉGÉ is due to the absence of adaptation capabilities in the generated interfaces. Adaptation is the process of varying the contents and behavior of an interface based on the characteristics of the interaction. Whereas adaptation in interfaces is usually related to the characteristics of the user, the features of the task and domain of the interaction are relevant as well [Puerta, 1990; Rissland, 1984]. To be able to generate interfaces to knowledge editors without knowing a priori the problem-solving method used for task modeling, PROTÉGÉ-II must provide the means for the generated interfaces to adapt to the task and domain of interest.

In this paper, we present the implementation of the method-independent PROTÉGÉ-II shell. In describing the system, we show the techniques applied to overcome the limitations of the original version of PROTÉGÉ. In particular, we show how the construction of problem-solving methods from mechanisms can be achieved using a librarylike storage–retrieval system and a graphical editing environment. We also describe the use of two formal languages developed to

provide adaptability in the interfaces generated by PROTÉGÉ-II. We claim that these languages provide independence from task and domain considerations in the interface-generation process, and give the knowledge engineer freedom to select specific presentation styles for each type of knowledge to be captured in the knowledge editors. Through the use of a user-interface management system (UIMS), PROTÉGÉ-II provides consistency in the interaction with the user throughout the various phases of knowledge-editor generation, and coordinates the complex relationships among interface components. Through the use of PROTÉGÉ-II, we hope to show how knowledge-acquisition tools can benefit from having access to multiple problem-solving methods and from presenting task-specific and domain-specific interfaces to the application expert.

The remainder of this paper is organized as follows. Section 2 introduces terminology necessary for the rest of the discussion. Section 3 offers an overview of the PROTÉGÉ-II system, describing the various subsystems in the shell and their role in the process of knowledge-editor generation. Section 4 presents an example of the use of PROTÉGÉ-II. Note that, since PROTÉGÉ-II is in a prototype stage, only part of the functionality described in Section 3 will be demonstrated in Section 4. To conclude, Section 5 discusses several research issues associated with the development and use of PROTÉGÉ-II.

2. Key Concepts

It is important that the concepts associated with the words *task* and *method* in PROTÉGÉ-II be specified clearly. In this section, we present these concepts along with definitions for the building blocks of the knowledge-level model of PROTÉGÉ-II (problem-solving mechanisms), and the two basic operations on these building blocks allowed by PROTÉGÉ-II: method configuration and method assembly. Given that the focus of this paper is on the architectural concept of PROTÉGÉ-II and its related human–computer interaction issues, our description of problem-solving mechanisms is intended to merely anchor the subsequent discussions. A more complete presentation of our views on methods and mechanisms has been presented elsewhere [Musen and Tu, 1991; Tu et al., 1991].

2.1. Tasks

A *task* in PROTÉGÉ-II is an activity, or an abstraction of an activity, in the real world. A task accepts some type of input and produces some type of output. The domain to which the task is applied determines the type of inputs accepted and the type of outputs produced.

Just as important as saying what a task is, is defining what a task is not. For our purposes, a task by itself is not a composite structure. Thus, it does not support a view of subtasks, or a decomposition into a hierarchy of subtasks. Such decompositions are possible, but only in the context of a problem-solving method, as shown is Section 2.3. In addition, a task does not impose a requirement on the type of knowledge needed to complete that task. This perspective is in contrast with the view of other researchers [Vanwelkenhuysen and Rademakers, 1990], but allows us to specify cleanly the functionality of PROTÉGÉ-II based on the manipulation of mechanisms and methods.

2.2. Mechanisms

A *mechanism* in our library is a procedure that completes, or solves, a task. It is a specification of *how* a task is accomplished. There is a many-to-many relationship between tasks and mechanisms that will serve as a base for defining the processes of method configuration and method assembly. The finite set of tasks that can be completed using a given mechanism is defined as that mechanism's *target;* the finite set of mechanisms that can complete a task is referred to as that task's *source*.

A mechanism imposes on its target tasks various requirements that determine what types of knowledge and data must be available to solve the task with that mechanism. In our implementation, these requirements derive from the components that make up a mechanism. These components are as follows [Musen and Tu, 1991]:

1. An input–output (I/O) declaration: The input declaration stipulates what inputs are required by the mechanism. The output declaration determines the target tasks of the mechanism.

2. A global data model: This component specifies the type of input data that the mechanism accepts and the type of output it produces, and the classes of operations that can be performed on the data.

3. A set of semantic constraints: This set defines relationships between inputs and outputs; thus, knowledge to verify the constraints must be accessible in the task.

4. A control- and data-flow configuration: The particular configuration used by a mechanism dictates that certain types of knowledge about the task be readily obtainable to allow the defined flow to operate. In other words, decisions that alter the flow of data or control may be dependent on certain knowledge about the task.

2.3. Methods

The union of the target tasks of all mechanisms in the PROTÉGÉ-II library neither is a complete set of all possible tasks, nor is intended to be one. There will be tasks that will fall outside the target of any of the existing mechanisms. It is possible, nevertheless, to assemble two or more mechanisms into a *method*. Mechanisms and methods hold the same many-to-many relationship with tasks. The target tasks of methods, however, are different in nature from the target tasks of mechanisms. Figure 1 illustrates this situation. So that we can differentiate the two types of tasks distinguishable when tasks are linked to methods or mechanisms, we call those that are part of a mechanism's target *simple* tasks, and call those belonging to a method's target *composite* tasks. A method in our library is a procedure that decomposes a composite task into subtasks, some of which may be composite as well. The decomposition process continues until all composite tasks and subtasks are reduced to simple subtasks. At this point, mechanisms can be applied to complete, or solve, the simple subtasks and, consequently, the original composite task. In addition to the requirements imposed on the tasks by the mechanisms making up a method, there may be requirements imposed on the composite task by the method itself. These requirements result from the assembly of the mechanisms into a procedure—the method—that has an overall data- and control-flow configuration.

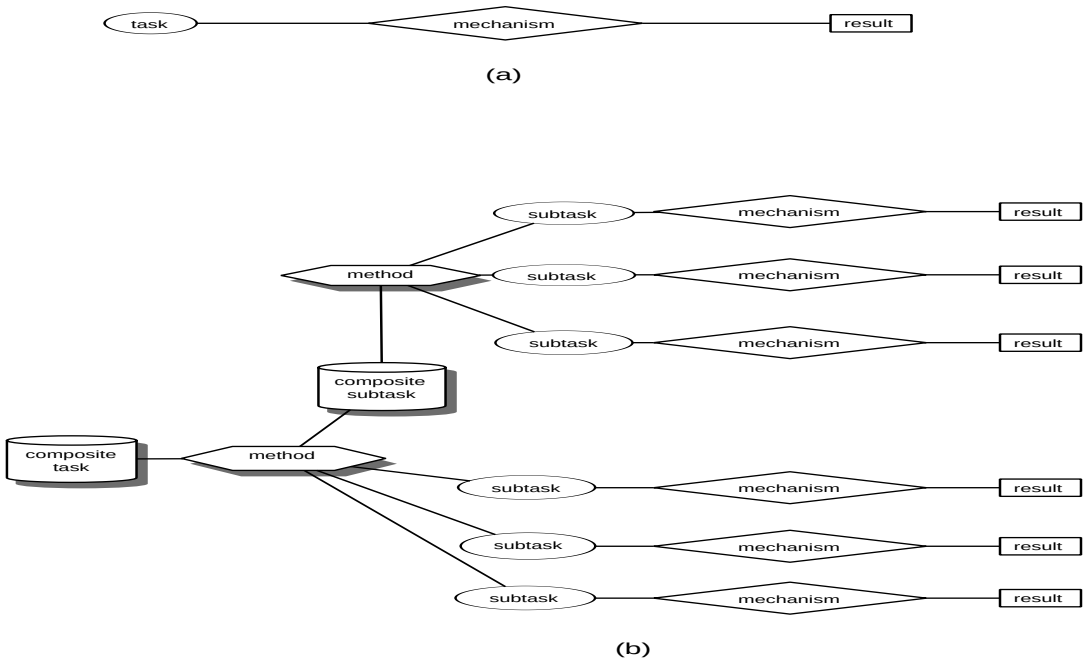

Figure 1. Methods, mechanisms, and tasks. In (a), a task is considered *simple* when it can be completed with a mechanism. In (b), a task is *composite* (noted by the shadowed outline) when it must be decomposed, perhaps recursively, into simple tasks, as imposed by a method, before a result can be obtained. A subtask is just a task that forms part of a task decomposition.

2.4. Method Configuration

The process of determining which mechanism, or method, should solve each subtask in a task decomposition is called *method configuration.* As shown in Figure 2, method configuration connects a task with a mechanism, or with a method, in the task's source. It is only when this connection is made that the task becomes composite or simple. If the task is connected to a mechanism, then it is considered *simple.* If it is connected to a method, then it is viewed as *composite* and must be broken into subtasks. After task decomposition is achieved, we complete the configuration process by configuring single mechanisms for each of the resulting simple subtasks.

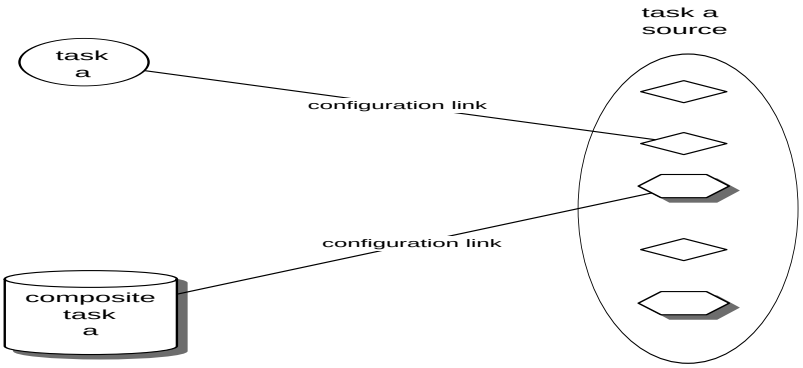

Figure 2. Method configuration. Configuring a method consists of establishing a link between a task and a mechanism, or a method, in the task's source. The same task can be viewed by PROTÉGÉ-II as simple or composite, depending on the link.

2.5. Method Assembly

The process of defining a new target by combining two or more mechanisms into a method is called *method assembly* and is illustrated in Figure 3. Putting together several mechanisms defines a task decomposition, which in turn determines a target of composite tasks fitting such decomposition.

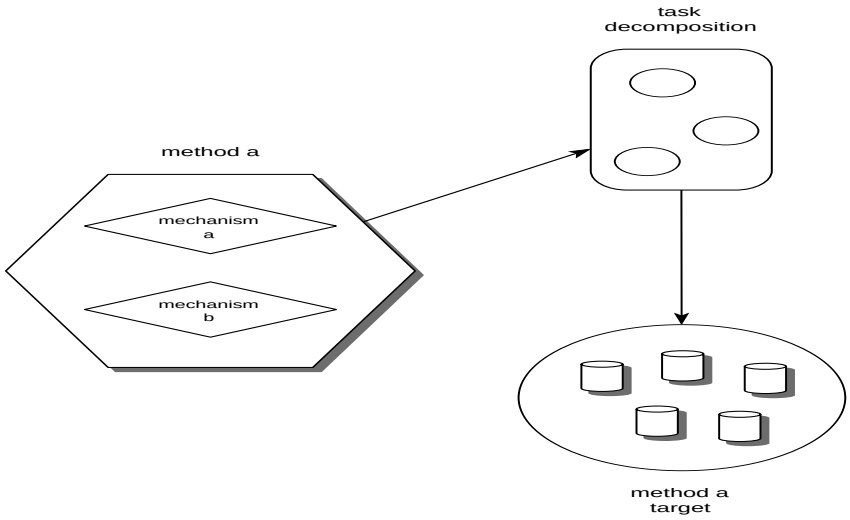

Figure 3. Method assembly. When a method is assembled through a combination of two or more mechanisms, a task decomposition for the method results. The task decomposition specifies how a composite task must be decomposed into subtasks so that the method can be used to solve the composite task. This task decomposition also defines a set of target composite tasks that can be decomposed in the manner dictated by the decomposition.

Note that the task decomposition resulting from the method assembly is what permits method configuration on the assembled method to take place at a later time. Although a method is assembled with individual mechanisms, PROTÉGÉ-II views each method by only that method's task decomposition. Therefore, at method-configuration time, PROTÉGÉ-II looks for the mechanisms that can complete the subtasks in the method's task decomposition. These mechanisms include, but are not limited to, the ones with which the method was assembled originally.

3. The PROTÉGÉ-II Shell

PROTÉGÉ-II is a general-purpose knowledge-acquisition environment that assists knowledge engineers in creating specialized knowledge-acquisition tools (knowledge editors) that are used by application experts to enter knowledge

into a knowledge base. Thus, PROTÉGÉ-II is considered a *metatool.* Unlike the original implementation of PROTÉGÉ, the reimplementation described in this paper does not require users to define knowledge in terms of a presupposed problem-solving method. Instead, users are given the ability to configure or assemble methods that will solve satisfactorily the class of tasks at hand.

Figure 4 shows an overview of the architecture of PROTÉGÉ-II. Knowledge engineers use the system to configure or assemble a method and, eventually, to develop a model of the task area for which knowledge is to be acquired. The system generates automatically a knowledge editor, which interacts with application experts to capture task-specific knowledge. A method- and domain-independent advice system uses the contents of the knowledge base and the given method to produce responses to its end users.

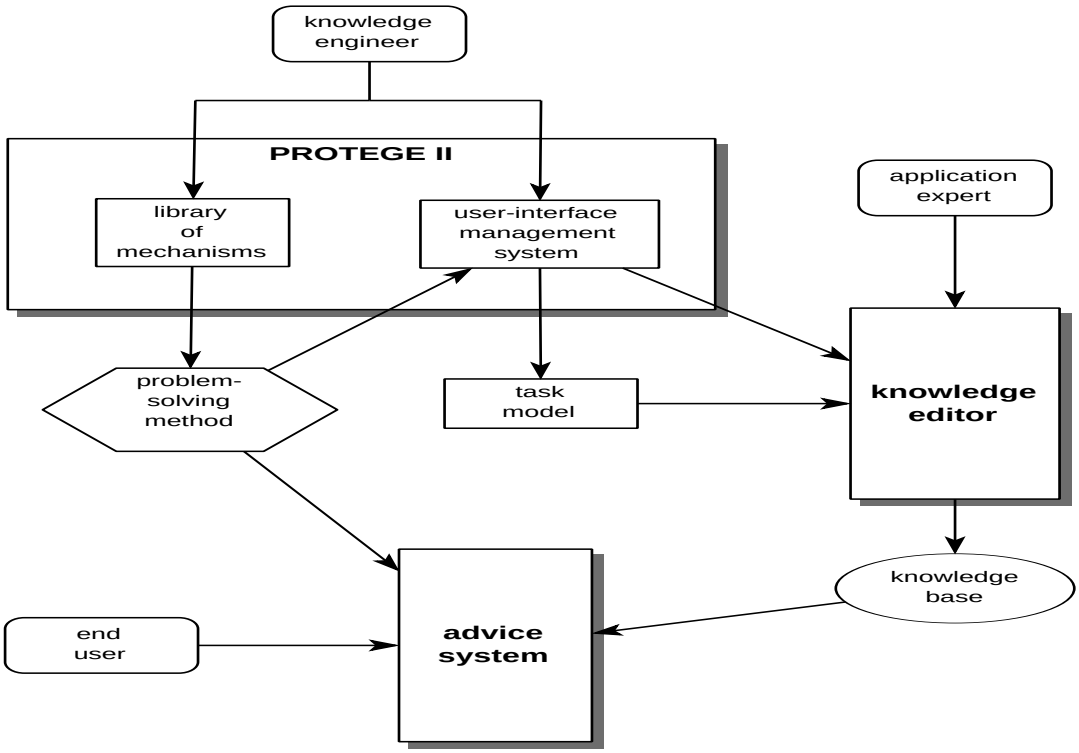

Figure 4. The PROTÉGÉ-II knowledge-acquisition shell. A knowledge engineer uses the facilities of the library of mechanisms to construct a problem-solving method. This method dictates how a model of the task is developed. Based on the task model, the knowledge engineer uses the user-interface management system to generate a knowledge editor. The application expert enters knowledge into a knowledge base using the editor. The knowledge base is ultimately utilized by an advice system that solves problems according to the constructed problem-solving method.

By providing configuring and assembling capabilities, PROTÉGÉ-II allows knowledge engineers to custom-tailor problem-solving methods for specific tasks. Furthermore, the resulting methods will have sufficient semantics for users to define at the knowledge level the roles that the entered knowledge will play in problem solving [Musen and Tu, 1991]. The previous PROTÉGÉ implementation forced users to define many of these roles at the symbol level.

3.1. General System Architecture

The architecture of PROTÉGÉ-II has two main subsystems (see Figure 4). Method assembly and configuration are conducted using the library of mechanisms. Task modeling and knowledge-editor generation are accomplished through the UIMS. This latter subsystem also defines the appearance and behavior of the generated interfaces, and the control of their various interface components. In Sections 3.2 through 3.4 these subsystems are described in further detail. The library of mechanisms was designed to overcome the limitations imposed in PROTÉGÉ by the assumption of a singular

problem-solving method. The UIMS was created to provide a high degree of freedom in the specification of the interaction style of the generated interfaces.

3.2. The Library of Mechanisms

The main function of the library of mechanisms is to provide the knowledge engineer with the facilities to look up mechanisms and methods, and to configure, or assemble, these methods in order to custom-tailor a problem-solving method for the task area at hand. In Figure 5, this subsystem has been further decomposed. As would be expected in any library, the index system is the central component. The index provides the means for organizing the search and selection of mechanisms. Depending on the mode (configuration or assembly) that the user selects, the index system presents either a configuration palette or a graphical assembly editor, respectively, to the knowledge engineer. The palette and the editor allow the necessary tailoring of candidate methods to produce the final problem-solving method. Note that the resulting method is, at this point, still domain-independent. The knowledge engineer will specialize this method for the particular domain during the phase of knowledge-editor generation.

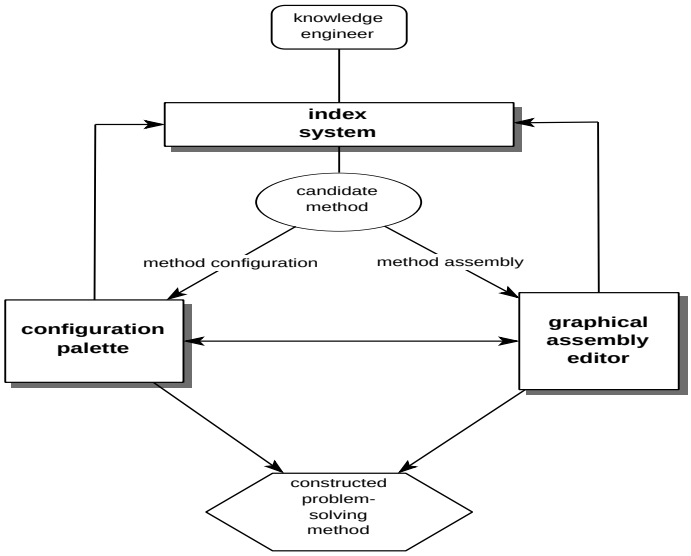

Figure 5. The library of mechanisms. Knowledge engineers search for candidate methods using the index system, which provides multiple search strategies. Candidate methods can be configured or assembled to obtain a final method that fits the task at hand. Typically, the process will require several iterations among the index system, the configuration palette, and the graphical assembly editor.

The underlying philosophy in the implementation of the index system is to give power to the knowledge engineer. We are not ready to remove the knowledge engineer from the cycle of expert-system construction. Instead, we want to present information to this user in a way that takes maximum advantage of the engineer's skills for selecting a suitable problem-solving method. This job can be equated with a student's preparation of a report on a given subject. The student may employ book-library facilities to find an already-written report on the subject, and may change, or supplement, some sections in that report using additional sources (giving proper credit, of course!). This action is the equivalent of method configuration. The student could also construct a new report by combining material from several sources in a process similar to method assembly. The library of mechanisms in PROTÉGÉ-II assumes that the knowledge engineer has the ability to judge the applicability and compatibility of methods and mechanisms, and has the skills to construct a problem-solving method using the available building blocks. In the same way that a library cannot prepare a report for a student, PROTÉGÉ-II will not, at the touch of a button, prepare a solution for the knowledge engineer's problem.

Indexing of mechanisms in PROTÉGÉ-II is based on the components of each mechanism and on its target tasks. There are three indexing strategies that we have identified as useful: by task, by global data model, and by I/O declaration.

The strategies complement one another, and use of one for a portion of the method configuration or assembly process does not preclude use of any other as a supplement.

3.2.1. Indexing by Task

Each mechanism, or method, is associated in the library with a set of tasks, or with a set of composite tasks, respectively. There are two ways in which this association can take place. First, when the mechanism is initially incorporated into the library, tasks can be specified to be associated with the mechanism. Second, during the normal use of the library, the knowledge engineer can establish that a mechanism is applicable for a task, and this information can be added to the library. When a search by task is conducted and a given task is selected, the index system lists all the mechanisms and methods associated with the task. Notice that this indexing scheme does not define a hierarchy of tasks.

Browsing through a list of tasks in the library of mechanisms has implicit limitations. One shortcoming is that the task's name as given by PROTÉGÉ-II, or the textual description for that name, may not exactly match the name and description that the PROTÉGÉ-II user is trying to find. In PROTÉGÉ-II, perfect matching is neither sought nor assumed likely. Instead, the general notion of custom-tailoring is applied, and the user is expected to navigate the process of constructing the problem-solving method. Consequently, the multiple-indexing capabilities of the library are of importance to facilitate the construction operation.

3.2.2. Indexing by Global Data Model

A mechanism specifies the type of data structures that it can accept through its data model. The model is global in the sense that it provides the base types with which PROTÉGÉ-II users can define concepts in the task's domain and build a task model from which a knowledge editor can be generated. The data model imposes limitations on the applicability of a given mechanism; thus, the data model can be used as part of the search criteria.

The index system can display a list of known global data models, and, on selection of one of these, can display the mechanisms and methods that use the data model. As with task indexing, no hierarchy of data models is constructed.

3.2.3. Indexing by Input–Output Declaration

The inputs and outputs that a mechanism requires also constitute a limiting factor in mechanism applicability. A mechanism, even though it may belong to a particular task's source, may not be able to complete that task in a given instance because all the inputs that it needs are not available. Furthermore, its outputs may not be as complete as desired, or may be insufficient to be assembled with another mechanism. The listing of inputs or outputs in browsers by the index system is similar to that of the other indexing strategies.

This strategy, when combined with the other two, affords the knowledge engineer multiple possibilities to search for candidate mechanisms and methods. This ability is important because we cannot determine a priori which indexing scheme constitutes the best limiting factor in a mechanism search. If, after employing the index system, the choice is a method, the user can configure, or assemble, this method using a configuration palette and an assembly editor.

3.2.4. The Palette and the Editor

The configuration palette and the graphical assembly editor are the tools that the knowledge engineer applies to construct a final problem-solving method that is custom-tailored for the given task. The graphical assembly editor allows the user to manipulate directly graphical objects to assemble new methods. It is more flexible than the palette, which simply permits the substitution of one of the method components for another. There is no restriction on using both tools simultaneously to, for example, configure a method by assembling a new method to fit as one of the configured method's components.

Our description of these subsystems is included with the discussion on the library of mechanisms because these functions are relevant in this context. The actual display and management of the subsystems' interfaces is conducted by the UIMS. This major subsystem is described in Sections 3.3 and 3.4.

3.3. The User-Interface Management System

Once a problem-solving method has been tailored, the knowledge engineer has to model the task area under the terms dictated by the method. Based on the task model, PROTÉGÉ-II then generates a knowledge editor whereby the

application expert can enter knowledge into the knowledge base. Both task modeling and knowledge editing are highly interactive processes in PROTÉGÉ-II. The interactive nature of the system places great demands in the definition and specification of the interfaces used by the knowledge engineer and by the application expert. This circumstance, coupled with the stated requirement of providing flexible interaction styles and the fact that the knowledge-editor interfaces must be generated automatically, creates a complex environment for interface specification. So that we can control this environment, we are developing a UIMS that permits a straightforward selection of interaction styles for different types of knowledge, and that directs the generation and display of these interfaces. Using the functionality of the UIMS, a knowledge engineer can *adapt* an interface to the characteristics of a task, a domain, and a method.

Interactions between PROTÉGÉ-II and knowledge-editor users take place through the use of forms or through the manipulation of graphical objects in an editor (Figure 6). The forms and graphical-objects compilers take programmatic textual descriptions written in a formal language, and produce specialized interfaces that guide the knowledge-acquisition routine by enforcing the domain-specific constraints specified through this formal language. The output of the forms compiler is a hierarchy of forms; the output of the graphical-objects compiler is an editor palette composed of graphical objects that can be combined in the editor according to a given editing paradigm (e.g., nodes-and-links, jigsaw puzzle). The current implementation of PROTÉGÉ-II supports only nodes-and-links graphs.

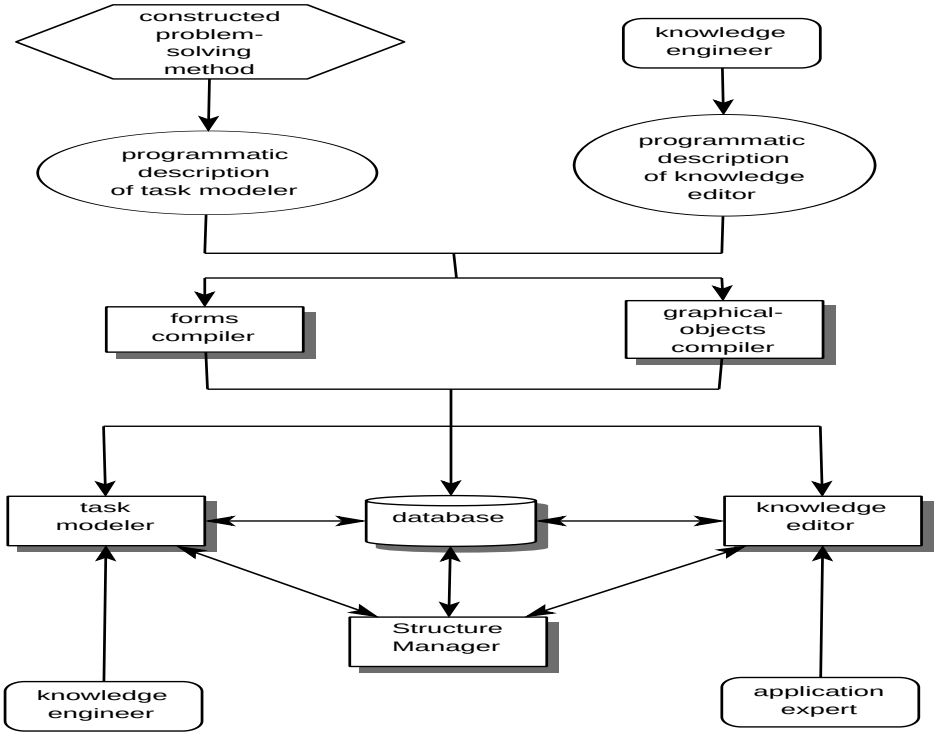

Figure 6. The user-interface management system. Programmatic descriptions of the task modeler and knowledge editor are processed by compilers that generate the necessary interfaces. A structure manager coordinates the display of all interface components using information stored in a common database.

Associated by design with each problem-solving method are programmatic descriptions of forms and of graphical objects that determine how the knowledge engineer can model the current task according to the chosen method, and how the domain-specific terms in the task can be defined. The association is made at method-entry time in the library of mechanisms, and the textual descriptions are a necessary part of each method's definition. Links between graphical objects and forms are maintained through the structure manager of the UIMS. This subsystem coordinates the interdependencies among forms and graphical objects, and manages the display of all structures according to these dependencies during task-modeling and knowledge-editing sessions.

To generate a new knowledge editor, the knowledge engineer makes use of the structures (forms and graphical objects) associated with the problem-solving method to enter the domain-specific terminology needed to model the task. These structures are presented to the knowledge engineer in the task modeler (see Figure 6). After building a model of the task, the knowledge engineer selects interaction styles for the various types of knowledge (e.g., nodes-and-links graphs for procedural knowledge) and PROTÉGÉ-II generates a *preliminary* knowledge editor based on those selections. Since this knowledge editor has been generated from a problem-solving method that is domain-independent, the knowledge editor is not yet adapted to the needs and requirements of the user (the domain expert), or to the particular task and domain. The adaptation of the knowledge editor is accomplished by the knowledge engineer who customizing the visual appearance of the graphical objects, and who specifies constraints, using formal languages, that will guide the knowledge-acquisition process. Finally, the knowledge engineer employs the facilities of the structure manager to stipulate the display relationships among forms and graphical objects. Using the structure manager, the knowledge engineer ultimately specifies how the inputs required by the problem-solving method relate to the knowledge entered in the knowledge editor. The editors generated by PROTÉGÉ-II are similar to that of OPAL [Musen et al., 1987]. There is an implicit assumption that the use of formal languages to generate interfaces provides enough flexibility to accommodate the varying requirements of the methods in the library. In Sections 3.3.1 through 3.3.2 we take a closer look at what is involved in the knowledge engineer's job of editing programmatic descriptions for the forms and graphical-objects compilers, and at how these descriptions shape the generation of the knowledge editors. The features described in these sections are exemplified in Section 4, where the use of PROTÉGÉ-II is demonstrated.

3.3.1. The Forms Compiler

The form-based interfaces used by the application expert, and those associated with problem-solving methods, are defined in a programming language, called FormIKA [Bennett, 1990], which was developed specifically for PROTÉGÉ-II. A program in this language has three goals:

1. Definition of variable-sized forms with multiple columns, value-entry fields, and value-selecting buttons
2. Definition of relationships among forms by creation of a hierarchy of forms in the interface
3. Definition of relationships among entries in the forms constituting a constraint system for the interface

An important part of the FormIKA language is the constraint-specification constructs. Whereas many user interfaces maintain relationships among interface objects by applying procedural attachments, FormIKA defines such relationships declaratively. Thus, knowledge engineers can use this language to declare domain-specific constraints that apply to particular pieces of knowledge that are acquired through the hierarchy of forms. In this manner, the constraints are used to guide the knowledge-acquisition session with the application expert and to adapt the knowledge-acquisition tool to the particular task, domain, and user. The same language is used by PROTÉGÉ-II to generate forms for the preliminary knowledge editor. These preliminary forms will have some constraints already specified that are generated from the model of the task constructed in the task modeler.

Constraints in FormIKA can be visual or nonvisual. Visual constraints deal with the appearance of entry blanks in the forms. Based on the value entered by the application expert in a particular entry blank, the constraint system can make other entry blanks accessible (visible), or inaccessible (nonvisible), thus directing the attention of the expert to the appropriate blanks. Nonvisual constraints work in a similar way, but affect instead the values displayed on the blanks themselves. Also, the constraint system can update or determine values in some entry blanks by applying defined constraints that relate to other entry blanks whose values have been entered already. In this manner, the consistency of the acquired knowledge is maintained.

3.3.2. The Graphical-objects Compiler

Both task modeling and knowledge editing can take place in graphical editors through manipulation of graphical objects (e.g., drawing a flowchart). These objects and their behavior can be specified through a formal language called Palette Constraint Language (PCL). Programs in this language have the following purposes:

1. Definition of a number of graphical objects that constitute a *palette* for a graphical editor
2. Definition of internal constraints that determine the behavior of the objects on the screen
3. Definition of external constraints that determine the connectivity of the graphical objects when these are used in a graphical editor

Again, the purpose of this language is to permit the definition of the constraints in a declarative fashion. The constraints have the effect of adding domain-specific behavior to otherwise general graphical editing structures, thus steering the interaction with the application expert.

Internal constraints limit the editing functions applicable to particular objects. For example, some objects can be declared as modifiable, giving the knowledge-editor user the ability to define subtypes of that particular object that inherit the parent's behavior. This feature is useful in instances where the task model defined by the PROTÉGÉ-II user is incomplete or is not sufficiently specialized, requiring the expert to define new terms derived from the more general task model.

External constraints put restrictions on the ability to connect one object to another. This type of constraints also is domain-dependent, and, together with the internal constraints, defines a behavior for the knowledge editor that will be generated. As noted before, PROTÉGÉ-II currently generates editors with a nodes-and-links editing paradigm. We are generalizing the grammars of FormIKA and PCL to form a single language that can support multiple views (forms, graphs) and multiple editing paradigms.

3.3.3. The structure manager

The on-screen display and handling of forms and graphical objects is accomplished with the structure manager. This subsystem can be used to bring up a specific interface structure, such as a form or a graphical object, or to associate one structure with another. The PROTÉGÉ-II user can relate the display of a particular form dynamically by double-clicking on an individual graphical object. The user can also associate an entry blank in a form with an object, or the label of a graphical object with a form. The structure manager also serves the vital role of linking the library of mechanisms with the task modeler and the knowledge editor. The special implications of this link are detailed in Section 3.4.

3.4. Communication Among Subsystems

Up to this point, we have described the library of mechanisms and the UIMS as two completely separate entities. The generation of a knowledge editor has been represented as a step-by-step process that moves from one subsystem to the next in an orderly manner. It is also possible, however, to access the library of mechanisms in PROTÉGÉ-II during the phases of task-modeling and knowledge-editor generation.

This link enables the knowledge engineer to define at the knowledge level the roles played by domain-specific control knowledge. For example, in the domain of hypertension, the structure manager can make the necessary links so that a drug dosage can be calculated using a specific mechanism (or set of mechanisms) from the library. Relationships between graphical objects and the library can be prescribed in similar fashion.

By means of this functionality, the PROTÉGÉ-II user can ultimately connect particular inputs of a method to forms, entry blanks, or graphical objects in the knowledge editor or in the task modeler. The definition of the roles of pieces of knowledge at the knowledge level was not always possible with the previous implementation of PROTÉGÉ [Musen and Tu, 1991]. In that system, many of these roles had to be defined at the symbol level by the knowledge engineer either as production rules or as procedures. By not presupposing a predefined problem-solving method, and by providing a library of mechanisms with which knowledge engineers can craft new problem-solving methods, PROTÉGÉ-II permits a complete definition of all knowledge roles through the tool itself.

4. A Running Example

Now that we are familiar with the different pieces that make up the PROTÉGÉ-II system, we shall illustrate the systems's use through an example of the generation of a knowledge editor. Because PROTÉGÉ-II is not fully implemented, not all the functionality outlined in the previous sections will be demonstrated in this example.

The knowledge engineer's goal in our example will be to produce a knowledge editor for a knowledge base that will be employed by an advice system that assists physicians managing cancer therapy. The task area is that of clinical-trial management in which patients are assigned at random to one of several alternative treatment plans. The details of a treatment plan are specified by physicians through what is called a *protocol*. Knowledge acquisition in this context consists of obtaining the description of protocols, which includes knowledge about the components of the protocols, the relationships among the components, and the procedural knowledge necessary to carry out the treatment plan.

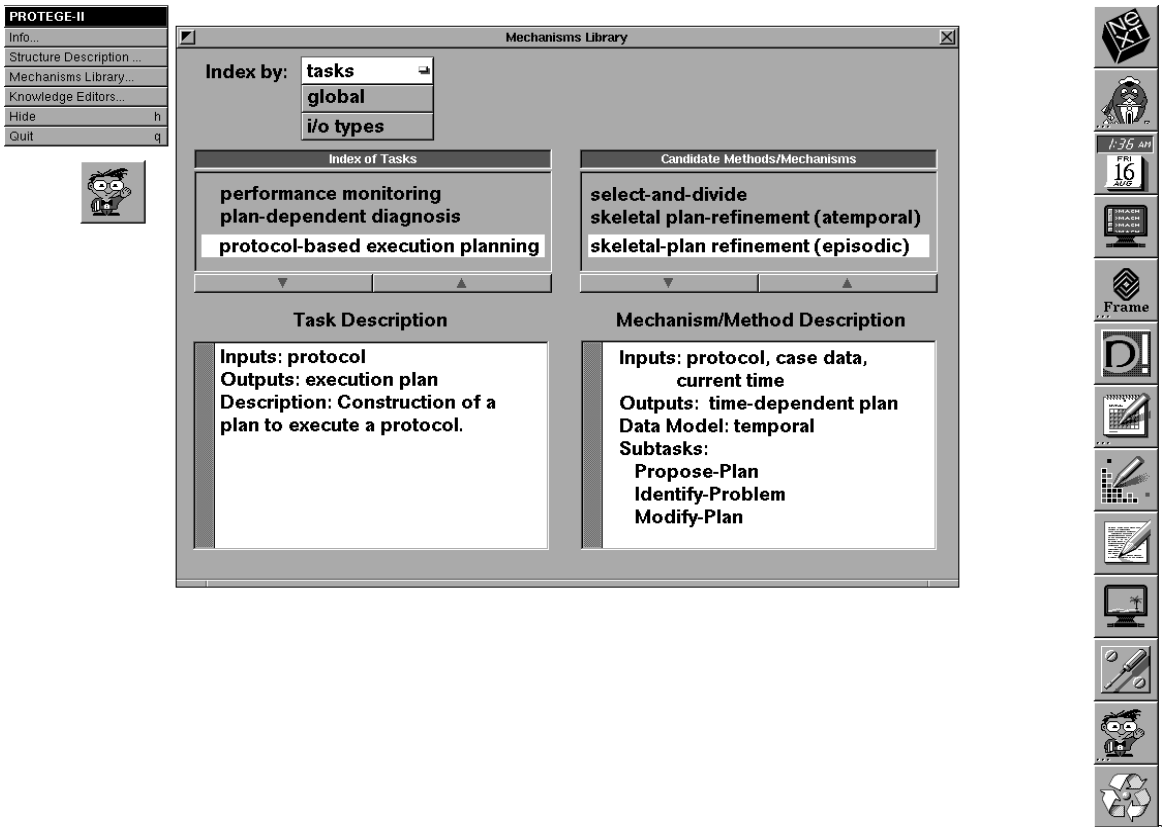

Figure 7. Interaction with the index system. The goal of the knowledge engineer is to find candidate methods or mechanisms that can be configured or assembled into a problem-solving method for the selected task.

4.1. Construction of the Problem-Solving Method

In the initial phase in the construction of the knowledge editor, the knowledge engineer uses the facilities of the library of mechanisms to look up candidate mechanisms, or methods, to solve the class of tasks at hand. Figure 7 shows the index system. The user has selected the index-by-task search strategy, and has found a suitable entry by browsing through a list of tasks. Selecting a task in this list displays both a textual description of the task, and the task's source in a separate browser. Similarly, the selection of an entry in the second browser shows textually the characteristics of that entry.

The descriptions of tasks and mechanisms appearing under the browsers aid the knowledge engineering in judging the appropriateness of the task and the applicability of the mechanism. In Figure 7, the user believes that *protocol-based execution planning* is a good match, and is investigating the applicability of the three known members of that task's source. In the domain of interest, it is necessary to manipulate temporal data (e.g., laboratory-test results, patient symptoms); thus, it is important that the mechanism selected support this type of data in its global data model. In this case, *episodic skeletal-plan refinement* is the only candidate method that fits this requirement (this information can be found in the textual descriptions). Additional details about a global data model can be extracted in the index system using the index-by-data-model search strategy.

Episodic skeletal-plan refinement is a method, as can be seen from the *task decomposition* associated with it. The user can now choose whether to configure this method in the configuration palette, or to call the assembly editor to modify the method's assembly. Figure 8 shows the configuration palette for our example. Each of the subtasks in the task decomposition is related to a multiple-selection button. The choices on the buttons are the mechanisms known to be in the subtask's source. The user can access information about these mechanisms in the index system by selecting the

corresponding subtask. Substituting one mechanism for another is as simple as clicking on the proper button. However, the user also is given the option of replacing a mechanism on the palette by one not present in the buttons.

For comparison, Figure 8 also depicts the same method in the assembly editor. Notice that the assembly editor is much more flexible because, by deleting or adding mechanisms, the user can define a different task decomposition. In addition, the editor allows the user to modify the control configuration of the method dictated by the connecting links between mechanisms. The PROTÉGÉ-II user would normally move back and forth among the palette, the editor, and the index system to construct the problem-solving method. If, by using the building blocks in the library of mechanisms, the user can custom-tailor a new method, this method will be saved and incorporated into the library for future use.

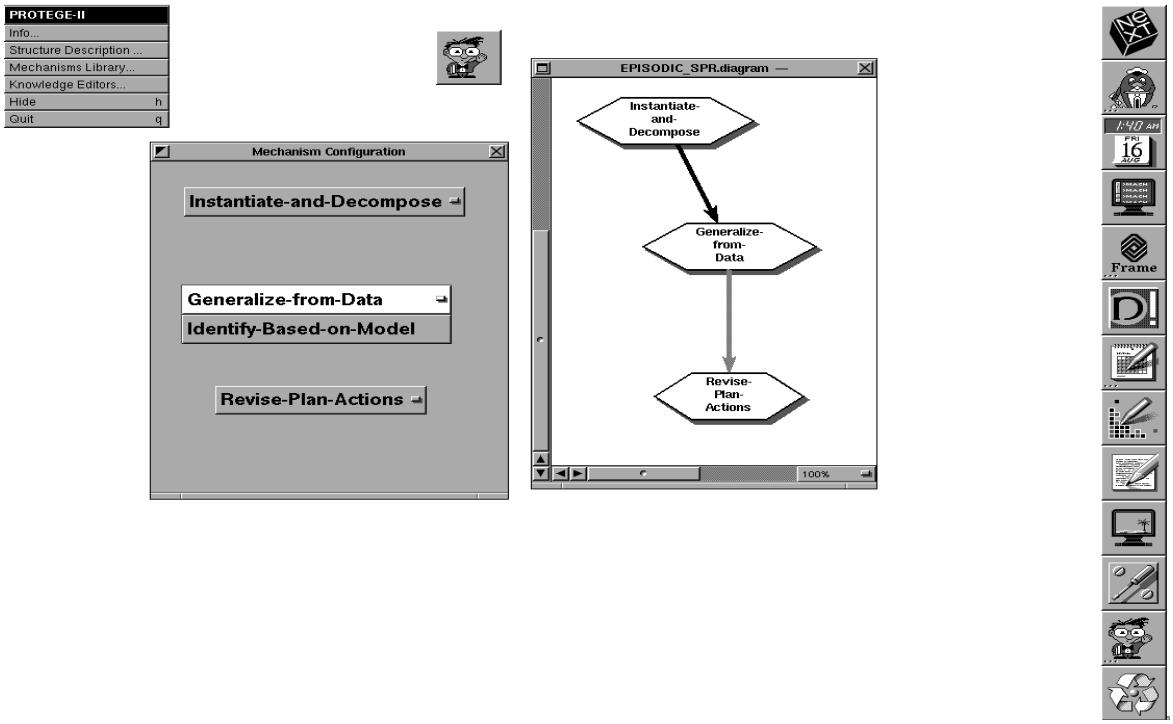

Figure 8. The configuration palette and the graphical assembly editor. The knowledge engineer can use the palette to select one of the available mechanisms for each of the subtasks in the method's task decomposition. Selecting a different mechanism using the index system, or assembling one with the assembly editor, is also possible. Notice that the assembly editor permits editing of the links between mechanisms, thus offering a level of flexibility much higher than that of the configuration palette.

4.2. Modeling of the Task

The problem-solving method of skeletal-plan refinement starts with the creation of an abstract (skeletal) solution to a problem. The abstract plan is decomposed into one or more constituent plans that are developed in more detail than is present in the original abstract plan. The constituent plans may, in turn, need to be decomposed further in a similar fashion. The process continues until a complete, specific plan for the given problem is defined. Episodic skeletal-plan refinement is a version of this problem-solving method that can operate with time-dependent data such as those in our running example.

The problem-solving method that is constructed using the library of mechanisms dictates the terms in which the task can be modeled. In our example, the task can be modeled in terms of *planning entities*, *input data*, and *actions*, as determined by the model of episodic skeletal-plan refinement [Musen and Tu, 1991]. Planning entities are time-varying 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. 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, and can decompose such entities into other planning entities, such as chemotherapies[1] and radiation therapies.

A laboratory-test result will be part of the input data, whereas drug attenuation can be an action that affects chemotherapy planning entities.

In addition, planning entities and 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.

As explained in Section 3.3, problem-solving methods in the library of mechanisms are associated with programmatic descriptions of forms and graphical objects. The UIMS processes these descriptions to display an interface for the knowledge engineer where the task-modeling interaction can take place. In Figure 9, we can see part of the interface for episodic skeletal-plan refinement. The form shown and others in the form hierarchy permit the knowledge engineer to define the domain-specific concepts for the given task as allowed by the problem-solving method. It is in this manner that construction of a method influences the task-modeling phase.

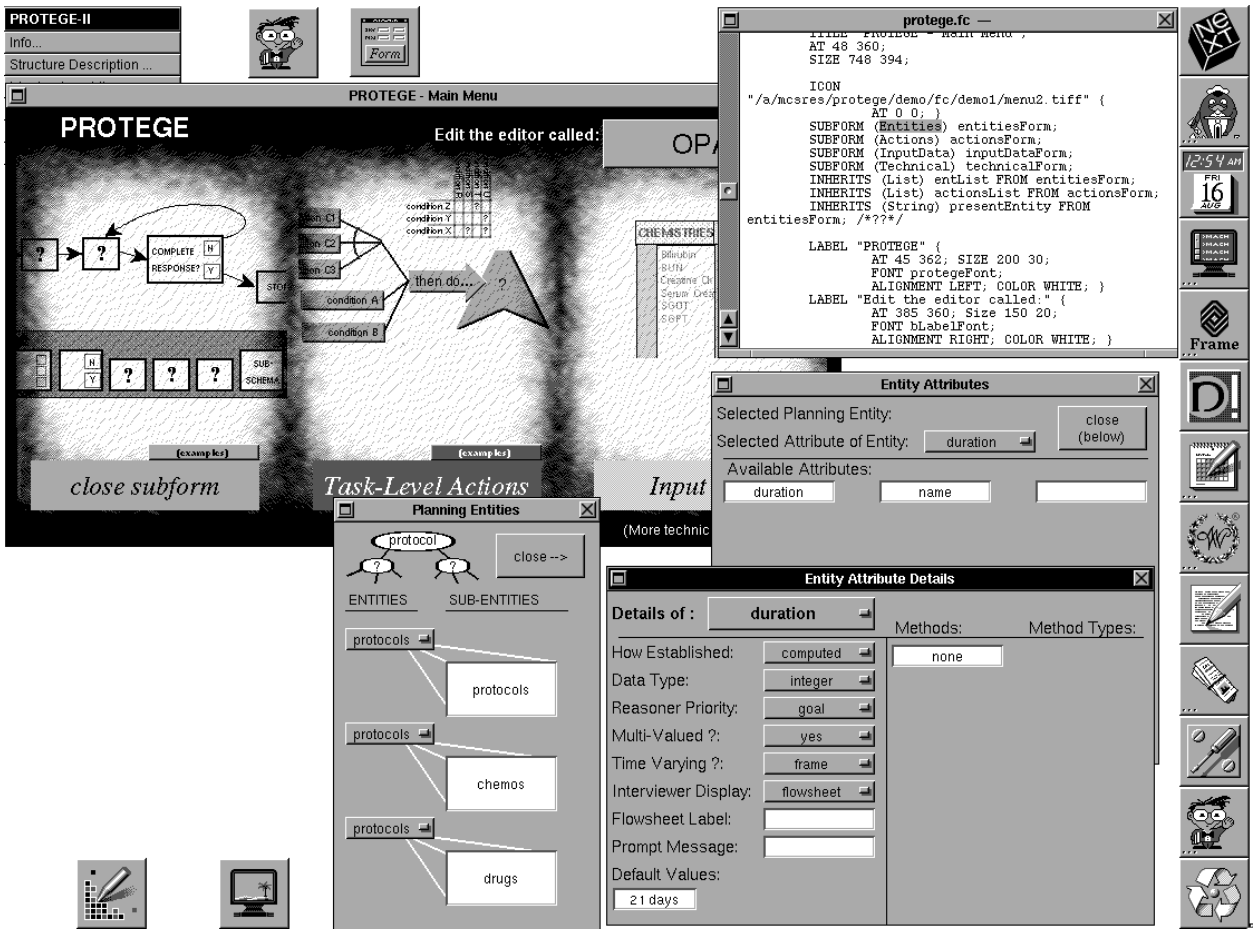

Figure 9. The interface for the task modeler. Each method is associated by design with an interface for modeling tasks in the terms prescribed by such method. The interfaces are specified using formal languages. This example shows forms to define planning entities, and attributes, for episodic skeletal-plan refinement, along with the FormIKA programmatic description of the forms.

1. Chemotherapies are groups of drugs administered to cancer patients in a particular order and in specific dosages.

One of the advantages of providing a UIMS is the resulting freedom in how the method requirements for task-modeling are enforced on the user. In this case, the knowledge engineer is presented with a number of forms where planning entities, attributes, actions, and input data can be defined. By associating the proper programmatic description of graphical objects with the problem-solving method, however, the user can construct the planning entity hierarchy in a graphical editor (e.g., as a tree). In fact, forms and graphical interfaces can both be associated with a given method and the user can make his own choice as to the interaction style to be employed.

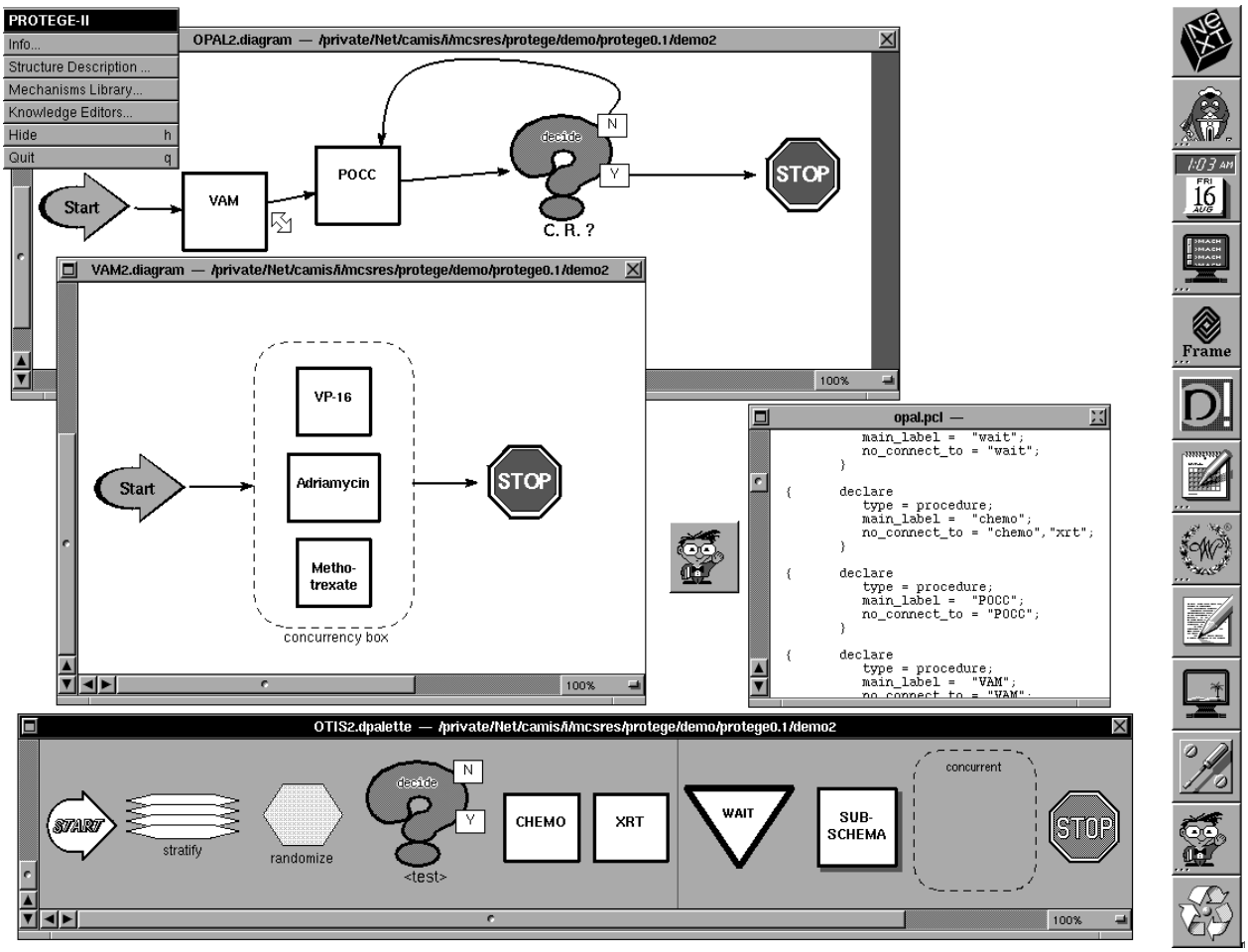

Figure 10. The graph-based portion of a knowledge editor to enter protocols. Application experts use a palette of graphical objects to construct flowcharts of their protocols. This example shows the palette, and a partial specification of a protocol, along with the associated PCL programmatic description of the palette objects. Because the knowledge engineer is allowed to stipulate the interaction style, the resulting palettes in PROTÉGÉ-II can be highly expressive, and closely related to the domain of interest.

4.3. Generation of the Knowledge Editor

The phase of knowledge-editor generation requires a specification of the interface between the application expert and the knowledge-acquisition system. Therefore, the knowledge engineer must define the contents of the interface, the presentation style of these contents, and the behavior of the generated interface components. Normally, knowledge-editor generation will comprise the following steps:

1. Determine the types of knowledge to be acquired
2. Decide the most appropriate style of presentation for each type of knowledge (form-based or graph-based)
3. Generate automatically a preliminary knowledge editor.

4. Write programmatic descriptions of the forms and graphical objects to be used, which constrains the behavior of the interface components

5. Define the connections between graphical objects and forms

In our continuing example, the knowledge about a protocol that will be acquired can be procedural or factual. The protocol is, by definition, a procedure. Clearly, however, many details about the protocol—such as types of drugs, and dosages of drugs—are factual. Since physicians typically draw flowcharts to provide high-level descriptions of protocols [Musen et al., 1987], it is appropriate to acquire this procedural knowledge graphically while obtaining facts about the protocol through the use of forms.

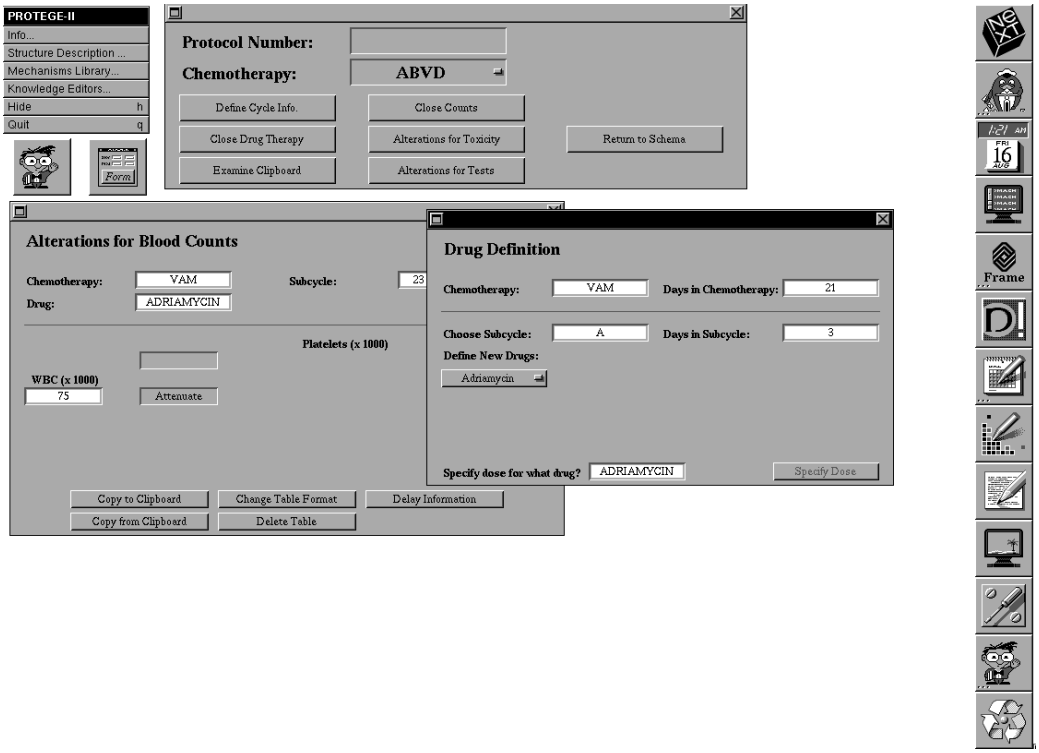

Figure 11. The form-based portion of a knowledge editor to enter protocols. This interaction style is preferred for certain types of knowledge, such as protocol data. The forms shown in this figure are part of a hierarchy of forms constructed by the knowledge engineer. The hierarchy controls the order in which the forms can be displayed, and thus guides the application expert through the knowledge-acquisition process.

Once the knowledge engineer makes a decision regarding presentation styles, a preliminary editor is generated by PROTÉGÉ-II from the task model developed and the presentation styles selected. This editor consists of a palette, or palettes, of graphical objects that can be utilized to draw diagrams in a graphical editor, and of a hierarchy of forms. The knowledge engineer then uses the forms and graphical-objects languages to impose domain-dependent constraints on the behavior of the interface components, thus adapting the editor to the requirements of the user, the task, and the domain. The PCL language, which constrains the behavior of the palette objects, predefines a number of types of graphical objects (e.g., iteration box, decision box) from which all other objects in the palette are derived. The predefined types provide a general framework to define graphical objects to capture procedural knowledge.

Figure 10 shows the palette for our example, as well as a partial protocol drawn using the objects in the palette. It also shows part of the PCL description for the palette. Notice that the declaration for the object *chemo* (for chemotherapy) has a constraint *no-connect-to* that determines that this object cannot be connected to itself. The graphical-objects compiler takes the PCL code and generates a knowledge-editor interface that automatically enforces the constraints specified in the PCL description. The compiler, however, does not prescribe a visual representation for the graphical objects that appear in the palette, although a default (not adapted) representation (where every object is a box

distinguishable by only its label) is available. In Figure 10, notice how the knowledge engineer makes use of this freedom to produce highly expressive palette objects that can aid the capture of knowledge from the application expert.

The protocol of Figure10 contains several examples of additional subtyping of graphical objects made by the application expert. For example, a subtype of the object *chemo* called *VAM* was defined. Subtyping of an object in the graphical editor is allowed unless a *no-modify* constraint is declared for the object.

The application expert enters certain facts about the protocols using forms. These forms are defined by the knowledge engineer, using the FormIKA language, as part of the process of knowledge-editor generation. Figure 11 shows some of the forms associated with the protocol in Figure 10 (Protocol 2091). Using FormIKA, the knowledge engineer defines a proper hierarchy of forms that guides the application expert during knowledge acquisition. Also, since automatic updating of entry fields can be defined through the language (e.g., the name of the protocol appears in the subforms automatically), the knowledge engineer can simplify the expert's work by designing forms such that there is no need to repeat entries.

The link between the protocol graph and the protocol form is maintained by the structure manager. This subsystem accesses a database that contains information on all interface structures and their directional links. Interface structures are added to the database from the FormIKA and PCL declaration of the structures. Links are added by the knowledge engineer. The structure manager has access as well to information about the constructed problem-solving method and its associated forms and graphical objects. The method itself is viewed by the structure manager as a graphical object that has inputs and outputs. Thus, to establish that a protocol is an input to episodic skeletal-plan refinement, the knowledge engineer requires no more than establishment of a link between that planning entity and the appropriate method input.

5. Discussion

For the past several years, researchers have been studying the use of models of problem-solving methods to construct knowledge-acquisition tools [McDermott, 1988; Musen, 1989a]. The resulting applications, although successful to some extent, have revealed serious limitations with this approach. The domain-independent nature of the models makes it difficult to express the role of knowledge that depends on domain-specific considerations. In addition, the resulting knowledge-acquisition tools are restricted to the class of tasks that the model can represent.

Whereas scientists have quickly recognized this problem, considerably less attention has been given to the human–computer interaction issues associated with the specification of interfaces for knowledge-acquisition tools built from models of problem-solving methods. A major difficulty in this area is the inability of interfaces derived from domain-independent models to express and capture knowledge that is task- and domain-specific, or to show any adaptability to the needs and requirements of the domain experts who use these tools.ðThese limitations are crucial because regardless of the appropriateness of a given model, a tool derived from such model is not useful if its interface is not suitable for communication with the user.

There is a clear paradox in the desire to build knowledge-acquisition tools that are based on domain-independent models but that simultaneously present interfaces that are task-, domain-, and user-dependent. Our approach to this challenge is two-pronged. On one hand, we are investigating the use of building blocks, called mechanisms, that can be used to construct problem-solving methods. By using mechanisms, we hope to eliminate the limitation of single-method architectures and to be able to define both domain-independent and domain-dependent knowledge roles. On the other hand, we are developing a user-interface management system that would allow us to adapt interfaces derived from the constructed problem-solving methods to the requirements of individual tasks, domains, and users. This capability is crucial in PROTÉGÉ-II, which is a metatool where interfaces for the resulting knowledge-acquisition tools are generated in great part automatically.

On the first front, our current endeavor is to identify mechanisms from expert systems previously developed in our laboratory [Tu et al., 1989; Musen, 1989a]. We are building the library of mechanisms as an effective architecture to index the identified mechanisms based on their components (see Section 3.2). The design of the library of mechanisms is similar to that of a library for books. The knowledge engineer can use multiple search strategies to find candidate mechanisms that can be further configured, or assembled, into a problem-solving method. Through the use of PROTÉGÉ-II, we expect to gain a better understanding of the indexing strategies that are most useful for knowledge engineers. Separately, we are studying the control problems that arise in the assembly and configuration of mechanisms and the graphical means to provide for the construction of problem-solving methods. The graphical assembly editor,

which is part of the library of mechanisms, will allow us to experiment with the data flow and control flow among mechanisms. We will investigate the extent to which the mechanisms in the library can be combined effectively in such an editor to form new problem-solving methods.

On the second front, we are examining means to generate adaptable interfaces in PROTÉGÉ-II, and are studying the effect of the adaptability requirement on the definition of mechanisms. One of our first results was the identification of an interface specification as a necessary component of a mechanism. In PROTÉGÉ-II, each mechanism defines what interface specifications can be used to model a task under the terms of such mechanism (see Section 4.2). In addition, the knowledge editors generated by PROTÉGÉ-II from the task model are considered preliminary. The user-interface management system provides the knowledge engineer with the means to adapt these preliminary editors to individual tasks, domains, and users. In this manner, we bridge the gap between an editor derived from a domain-independent model and an editor tailored to the requirements of a single task, domain, and user. Although the current version of PROTÉGÉ-II supports only form- and graph-based interaction modes, we are merging FormIKA and PCL—our formal languages for interface specification—into a single, more general, language that will support additional modalities and drawing techniques (e.g., jigsaw puzzle). We hope not only to gain additional flexibility in the generation of interfaces, but also to study approaches for mechanism assembly different from the nodes-and-links paradigm.

PROTÉGÉ-II's view of mechanisms as reusable programming constructs parallels, in certain aspects, that of Spark [Klinker et al., 1991]. Perhaps a clear contrast between these two systems lies not in the particular techniques used in the manipulation of mechanisms, but rather in the different approaches to the interaction between the user and the system. Whereas PROTÉGÉ-II aides a knowledge engineer in the construction of knowledge-acquisition tools, Spark guides a nonprogrammer through the process of selecting an assemby of mechanisms. PROTÉGÉ-II has the relative advantage of presenting information to a knowledgeable user who can make informed decisions. Spark, in contrast, relies heavily on the system to make critical choices.

One system that recognizes the central importance of domain-specific interfaces for knowledge-acquisition tools is DOTS [Eriksson, 1990]. In DOTS, knowledge engineers can construct interfaces for knowledge-acquisition tools by specifying a window-system layer on top of the Interlisp-D system. The process, however, is mostly manual. The intent of the user-interface management system in PROTÉGÉ-II is to relieve the knowledge engineer from having to specify in its entirety the interface for each knowledge-acquisition tool developed. The preliminary knowledge editor produced from the task model is fully usable and the adaptation of this preliminary editor is greatly facilitated by the existence of formal languages that allow the stipulation of constraints in a declarative fashion.

Our emphasis on the composition and features of the interfaces presented by PROTÉGÉ-II derives from our previous experience with PROTÉGÉ. In knowledge-acquisition tools, the nature of the interface—where knowledge is presented to and elicited from users—may be as important as the means used to model the class of tasks at hand. The adaptability of this communication channel to individual tasks, domains, and users has not been given sufficient attention in the research community and is one of the key aspects of PROTÉGÉ-II that we are studying. The challenge remains, for us as well as for other groups, to develop knowledge-level architectures that can generate knowledge-acquisition tools that do not overlook the vital link to the source of knowledge: the interface to the application expert.

Acknowledgments

This work has been supported in part by grant LM05157 from the National Library of Medicine, by grant HS06330 from the Agency for Health Care Policy and Research, by a grant from the Institute for Biological and Clinical Investigation at Stanford University, and by a gift from Digital Equipment Corporation. Computer support was provided in part by grant RR05353 from the Biomedical Research Support Grant Program of the National Institutes of Health, and by the SUMEX-AIM resource, supported by grant LM05208 from the National Library of Medicine.

We thank John Dawes, Josef Schreiner, Yuval Shahar, and James Winkles for their insightful comments, Andrew Bennett for his work in developing FormIKA, and Lyn Dupré for her extensive editing of a previous draft of this paper.

References

Bennett, J. S. 1985. ROGET: A knowledge-based system for acquiring the conceptual structure of a diagnostic expert system. *Journal of Automated Reasoning* **1**:49–74.

Bennett, A. 1990. *A form-based user interface management system for knowledge acquisition.* Master's Thesis. KSL-Report 90-43, Knowledge Systems Laboratory, Stanford University, Stanford, CA.

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

Clancey, W. J. 1985. Heuristic classification. *Artificial Intelligence* **27**:289–350.

Eriksson, H. 1990. Meta-tool support for customized domain-oriented knowledge acquisition. In *Proceedings of the Fifth Banff Knowledge-Acquisition for Knowledge-Based Systems Workshop,* Boose, J. H. and Gaines, B. R., editors., pp. 6.1–6.20. Banff, Alberta, Canada.

Friedland, P. E., and Iwasaki, Y. 1985. The concept and implementation of skeletal plans. *Journal of Automated Reasoning* **1**:161–208.

Klinker, G., Bhola, C., Dallemagne, G., Marques, D., and McDermott, J. 1991.Usable and reusable programming constructs. K*nowledge Acquisition* **3**:117-135.

Marcus, S. and McDermott, J. 1989. SALT: A knowledge acquisition tool for propose-and-revise systems. *Artificial Intelligence* **39**:1–37.

McDermott, J. 1988. Preliminary steps toward a taxonomy of problem-solving methods. In *Automating Knowledge Acquisition for Expert Systems,* ed. by Marcus S., pp. 225–256. Boston: Kluwer Academic.

Musen, M. A., Fagan, L. M., Combs, D. M., and Shortliffe, E. H. 1987. Use of a domain model to drive an interactive knowledge-editing tool. *International Journal of Man–Machine Studies* **26**:105–121.

Musen, M.A. 1989a. *Automated Generation of Model-Based Knowledge-Acquisition Tools.* London: Pitman.

Musen, M. A. 1989b. An editor for the conceptual models of interactive knowledge-acquisition tools. *International Journal of Man-Machine Studies* **31**:673–698.

Musen, M.A., and Tu, S.W. 1991. *A model of skeletal-plan refinement to generate task-specific knowledge acquisition tools.* KSL-Report 91–05, Knowledge System Laboratory, Stanford University, Stanford, CA.

Newell, A. 1982. The knowledge level. *Artificial Intelligence* **18**:87–127.

Puerta, A. 1990. L-CID: *a blackboard framework to experiment with self-adaptation in intelligent interfaces.* Ph.D. Dissertation. Technical Report USCMI 90-esl-6, Center for Machine Intelligence, University of South Carolina, Columbia, SC.

Rissland, E. 1984. Ingredients of intelligent interfaces. *International Journal of Man–Machine Studies* **21**:377–388.

Steels, L. 1990. Components of expertise. *AI Magazine* **11**(2):30–49.

Tu, S., Shahar, Y., Dawes, J.,Winkles, J., Puerta, A., and Musen, M. 1991. A problem-solving model for episodic skeletal-plan refinement. In *Proceedings of the Sixth Knowledge-Acquisition for Knowledge-Based Systems Workshop,* ed. by Boose, J. H. and Gaines, B. R., Alberta, Canada, in press.

Vanwelkenhuysen, J., and Rademakers, P. 1990. Mapping a knowledge level analysis onto a computational framework. *Proceedings of the Tenth European Conference on Artificial Intelligence,* pp. 661–664. Stockholm, Sweden.