A Problem-Solving Model for Episodic Skeletal-Plan Refinement

Samson Tu, Yuval Shahar, John Dawes, James Winkles, Angel Puerta, Mark Musen

Medical Computer Science Group

Knowledge Systems Laboratory

Departments of Medicine and Computer Science

Stanford University School of Medicine

Stanford, CA 94305-5479

Running title: Episodic Skeletal-Plan Refinement

Title: A Problem-Solving Model for Episodic Skeletal-Plan Refinement

Running title: Episodic Skeletal-Plan Refinement

Authors: Samson Tu, Yuval Shahar, John Dawes, James Winkles, Angel Puerta, Mark Musen

Please send all correspondences to:

Samson Tu
Section on Medical Informatics
2nd Floor, Medical School Office Building
Stanford University School of Medicine
Stanford, CA 94305-5479

ABSTRACT

PROTÉGÉ is a metalevel program that generates knowledge-acquisition tools that are based on the method of skeletal-plan refinement. In this paper, we propose a flexible and extensible architecture that allows the problem-solving method to be assembled from more basic methods. In this architecture, we emphasize (1) a uniform view of problem solving at different levels of granularity, (2) an explicit data model that allows construction of complex datatypes from predefined datatypes, and (3) the inclusion of domain-dependent control information within a domain-independent problem-solving method. We show how such a model of problem solving can drive the generation of knowledge-acquisition tools.

1 INTRODUCTION

Skeletal-plan refinement is a problem-solving method originally proposed for designing molecular-biology experiments (Friedland and Iwasaki, 1985). In the 1980s, we modified and extended the method for planning therapies for clinical trials (Tu et al., 1989). Because the method is used to instantiate skeletal plans at multiple time points, we call it the method of *episodic skeletal-plan refinement* (ESPR). The method was the basis for a series of experiments in building knowledge-acquisition systems. First, we built OPAL, a system for acquiring skeletal plans in the domain of cancer chemotherapy (Musen et al, 1987. Then, our laboratory developed PROTÉGÉ, a metalevel tool for generating knowledge-acquisition systems for clinical-trial management in alternative domains (Musen, 1989).

Like several other method-based knowledge-acquisition systems built at approximately the same time—ROGET for a type of heuristic classification (Bennett, 1985), MOLE for the cover-and-differentiate method (Eshelman, 1988), and SALT for the propose-and-revise method (Marcus and McDermott, 1989)—PROTÉGÉ assumes that there is a fixed

method of problem solving that defines the roles that domain knowledge plays in the problem-solving process. PROTÉGÉ, unlike the other systems, divides the knowledge-acquisition process into two phases. First, a knowledge engineer maps the terms and relations describing the method's knowledge roles into concepts in the domain. In that process, she creates, in domain terms, a model of the knowledge required for solving the task using this method. Using this task model, PROTÉGÉ generates a knowledge-acquisition tool specialized for the task in the domain. Because the terms used in it are meaningful in the domain, the knowledge-acquisition tool is intended for use by domain experts (who may be unfamiliar with the problem-solving method) to enter the detailed content knowledge. An inference engine called e-ONCOCIN interprets the resulting knowledge base to create an advice system (Figure 1).

(Insert Figure 1 here)

As formulated in PROTÉGÉ, the ESPR method uses a collection of time-varying *planning entities* to represent the operators that can be applied, *input data* that are gathered from the environment, and *task actions* that can modify the instantiation of currently active planning entities in response to particular input data. In the domain of cancer treatment, for example, the planning entities might be chemotherapies and radiation therapies; the input data might be results of laboratory tests and physical examinations; and task actions might be rules for reducing the medication doses based on observations of drug toxicities. These treatment options and rules for dose adjustments are often described in *protocols* that are followed by physicians. Thus, in treatment for a patient with cancer, the domain-specific knowledge-acquisition tool generated by PROTÉGÉ is a tool for acquiring protocols that specify both different types of chemotherapies and radiation therapies and the alternative ways of modifying the therapeutic actions in response to input data.

This methodology of using a fixed problem-solving method to define knowledge roles, although powerful, has a number of limitations. The method, being embodied in the code

of the e-ONCOCIN inference engine, is not easily extensible.When the domain task to be solved by the method is described, there may be domain-dependent attributes that do not fit into any of the predefined roles of the method. For example, in cancer chemotherapy, the administered chemotherapies are counted in terms of cycles. Thus, an oncologist can speak of “second cycle of VAM chemotherapy.” Yet the domain-independent ESPR method provides no guidance for defining such small-grained domain-dependent concepts. The knowledge engineer has to provide operational semantics for such domain-dependent terms by writing production rules in the inference engine’s rule language. A similar problem arises because there is no easy way, except by writing symbol-level production rules, to add data-abstraction behavior to the ESPR method that is presupposed by PROTÉGÉ. The source of power of a methodology can be a limitation when the problem to be solved does not fit exactly into the framework of the methodology. The separation of control knowledge in the problem-solving method and content knowledge in the domain allows us to build knowledge-acquisition tools. Nevertheless, there are tasks for which some specific types of domain-dependent control knowledge are needed. In cancer treatment, one example of such domain-dependent control knowledge involves the sequencing of multiple chemotherapies over time. In PROTÉGÉ and in the knowledge editors generated by PROTÉGÉ, we showed that it is possible to integrate such information in our knowledge-acquisition systems. Yet this possibility was not extended systematically to other parts of the problem-solving method, such as the ordering of dose-reduction task actions, where knowledge engineers have had to order the underlying production rules manually to obtain the right behavior.

Another problem in PROTÉGÉ—although not one that is an intrinsic weakness of the methodology—is that PROTÉGÉ’s data model does not permit the user to represent relationships between past input data and the planning entities that are modeled as temporal intervals. Thus, even though the representation language in the e-ONCOCIN problem solver allows the knowledge engineer to specify a temporal relationship such as “FIRST

white-blood-cell count AFTER LAST chemotherapy," such a reference cannot be entered through the knowledge editor.

These considerations have led us to design a new PROTÉGÉ architecture, called PROTÉGÉ-II, that is configurable and extensible by the users both of PROTÉGÉ and of the knowledge editors created by PROTÉGÉ. It allows knowledge engineers to develop knowledge editors that are based on problem-solving methods other than ESPR, and to enter domain-dependent control knowledge in constrained ways. Furthermore, it has a consistent data model and a uniform view of problem solving at different levels of granularity.

We shall describe our approach to developing a model of problem-solving behavior that satisfies these objectives (Section 2). We propose that knowledge engineers may be able to configure a model of problem solving from a library of subcomponents that can serve as building blocks for custom-tailored problem-solving methods at different levels of granularity. We shall reformulate the ESPR method as a composite method assembled from other methods (Section 3). Finally, we shall show how the two-stage knowledge-acquisition process is applied in the PROTÉGÉ-II architecture (Section 4). In this paper, we focus on the reformulation of the ESPR method in the PROTÉGÉ-II architecture. The design of PROTÉGÉ-II to implement the architecture, and issues of the related human–computer interaction, are described elsewhere (Puerta et al., 1992).

2 The PROTÉGÉ-II ARCHITECTURE

We adopt a framework where problems to be solved are represented as *tasks*, and ways of solving them are represented as *methods*. A task is a specification of a problem to be solved in the world, such as diagnosing problems in a four-cylinder engine or designing a waveform generator. A *task class* is a family of tasks whose input and output specifications can be characterized in similar domain-independent terms. For diagnostic problem

solving, the terms may be “symptoms” and “faults.” For a design problem, the terms may be those describing the functional specifications of the required output and the available input components. In PROTÉGÉ-II, these input and output specifications, together with a task index, relations that the inputs and outputs must satisfy, and an optional predicate indicating when the task is completed, characterize the task class. No assumption is made about the procedure or the knowledge that is required for accomplishing the task.

Abstracting a task class from tasks applied in different domains is useful when we can define problem-solving methods that construct solutions for tasks in the task class. A problem-solving method is a partially predefined procedural specification for transforming input data objects into output data objects according to transformation rules that have meaning at the knowledge level. The input data objects include not only the case data on which the problem solver is working, but also domain knowledge that plays definite roles in the transformation rules. In the ESPR method that we shall describe (Section 4), the inputs include the case data, the skeletal plan that has been selected for the case, and the current time of the session. The output of the method is a set of planning-entity instances that represent real-world actions executable by the user of the advice system.

A method either is decomposable into a set of subtasks or is a basic method, which we call a *mechanism*. Each of the subtasks may have one or more methods for its solution (Figure 2). This recursive decomposition of methods into subtasks gives us a uniform view of the problem-solving process at different levels of granularity. Note that the level to which we decompose a method is a design decision regarding the appropriate level of abstraction. What is considered a mechanism in one context may well be considered a method in another. Our guideline is to define mechanisms that are sufficiently specific to solve particular tasks and yet sufficiently general to be reusable as parts of other problem-solving methods.

A method specification is analogous to a task specification. In addition to input and output declarations and semantic-constraint annotations, it has the following components:

1. A global data model that specifies the type of data and knowledge processed by the method
2. A collection of subtasks, and relations specifying how the subtasks' inputs are to be bound from the inputs of the method
3. A control structure for the subtasks, embedded in the code of an interpreter for the method, or specified explicitly in the task-control language (for the purpose of this paper, the task-control language is simply a partial ordering among the subtasks)

(Insert Figure 2 here)

A method is partially predefined in the sense that the input and output data objects on which the method operates must be specialized into those that have meaning to the domain expert, and the methods or mechanisms for solving the method's subtasks must be selected and ordered. Because methods have different knowledge requirements, some methods may not be appropriate for configuring the problem solver in a particular domain. On the other hand, multiple methods may be appropriate for a single task. It is the knowledge engineer's responsibility to identify the appropriate methods for the tasks on which she is working. We call this process of data specialization and method selection and ordering *method configuration*.

PROTÉGÉ-II will have libraries of prebuilt task classes and methods. When appropriate, knowledge engineers will configure methods for tasks in particular domains. However, the domain task to be solved sometimes will not have an appropriate prebuilt problem-solving method. In this case, either there is nothing in the libraries that fits the task and the knowledge engineer will develop a new task description and its associated methods, or the knowledge engineer will *assemble* existing methods into composite methods.

To assemble composite methods from other methods, we must have a language for specifying terms and relations in a method. This method-specification language must have a *data model* for specifying the knowledge and data processed by the method, a *control model* for selecting and sequencing the execution of the subtasks or submethods in the problem-solving process, and a language for describing the semantics of the relationships among the inputs and outputs of the tasks and of the methods. The data model allows a knowledge engineer to declare, as input to the constructed task and method, datatypes that are composed from the datatypes associated with the existing tasks and methods. The data model also specifies the kinds of query and updates that can be performed on the data. The control model allows her to partition the required control information, some of which she specifies as she assembles the problem-solving method, while leaving other parts for the domain expert to enter.

An analogy to a programming language illustrates the need for these three aspects of the problem-solver definition. A programming language provides a set of basic datatypes (such as strings and structures) from which application data structures can be declared. Similarly, each problem-solving method must presuppose some general model of the data it processes. That data model provides the base datatype language with which knowledge engineers can define other concepts in the domain. A programming language has control structures such as *for* loops and *if–then* statements. At the level of the problem-solving process, we must specify the ordering of the invocation of the subtasks and their methods, either statically or dynamically. Finally, a programming language has formal and informal specifications of the semantics of its operations. We do not expect to be able to develop a language for formally specifying the behaviors of the problem solvers. However, with each task and method, we shall associate descriptive annotations on the properties of its inputs and outputs. With each method, we shall associate similar annotations on its knowledge requirements and expected problem-solving behavior.

3 COMPONENTS OF THE ESPR METHOD

We shall illustrate the architecture for PROTÉGÉ-II by describing in detail how we have reformulated the ESPR method in the new framework. Although the ESPR method was built into the original PROTÉGÉ, it can be assembled from those methods and mechanisms in the built-in method library of PROTÉGÉ-II. We shall describe first the global data model that is assumed throughout, then the partitioning of the control information into the component that is prebuilt into the methods, the component that is configured by the knowledge engineer, and the component that requires domain expertise. Finally, we shall describe the subtasks and the methods from which the ESPR method can be assembled.

3.1 Global Data Model

The nature of the cases that an expert system solves depends on the task and on the domain. Those cases, and the solutions to those cases, must be represented by a collection of data structures. Consequently, a problem-solving method such as episodic skeletal-plan refinement must presuppose some general model of the data it processes. That data model provides the base datatype language with which knowledge engineers can define other concepts in the domain. The global data model is a property of a class of domains and is not unique to any particular problem-solving method. We assume that each method presupposes a global data model whose primitives can be specialized—but not augmented—by the PROTÉGÉ-II user.

Time Model: A function of the global data model is to define the appropriate temporal properties of data. In the domain of medical therapy planning for which PROTÉGÉ was originally designed, patient data can be time-invariant (such as the patient's social security number), time-stamped (such as laboratory-test results), or interval-based (such as a treatment intervention that takes place over time). In the model of time for skeletal-plan refinement, we have adopted the time point as the primitive temporal element. We assume that

time points have granularities expressed as a calendar/clock time unit. The use of time points with granularities implies that we have adopted a model of discrete time points at each level of time granularity. We assume that the granularity units are chosen such that a time point specified at a finer level can always be converted to a coarse level. For example, “12-May-89” can be converted to “May-89.” In this model, intervals are defined as a pair of time points, and the semantics of the intervals is defined in terms of the corresponding time-point pairs.

There are three special time points: PAST, FUTURE, and NOW. For all time points *t*, *t* is after PAST and before FUTURE. NOW represents the “current time” (or today’s date, if date is the granularity of time) associated with a particular patient case in an interaction with the problem solver.

We define precedence relations over time points, granularities, and durations. These precedence relations must take into account granularities of the time stamp. For example, we define two types of equalities between time points with different granularities. In one equality relation (≈), the two time points are compared at their common granularity (thus, “1990 ≈ June 1990”); the other equality relation (same_time) requires that time points be compared at their finer granularity (thus, “1990 same_time June 1990” is not true). For time intervals, we have defined corresponding relations based on the relations on their start and stop points.

In a time model for skeletal-plan refinement, assertions are interpreted over time points. Thus, it is possible to ask whether something is true at a particular time. If an assertion is associated with an interval, then that assertion holds for all time points in the interval.

Entities: In the data model that we are using for our reformulation of ESPR, the primitive data elements are entities with attributes. The entities are class objects that may have instances; the attributes of the class objects may be either class attributes whose values are

common to all instances or instance attributes whose values must be determined for each instance.

We want to be able to define basic relations among the class objects, such as the *kind of* relation (e.g., a prescription for AZT drug is a *kind of* medication, which is a *kind of* planning entity), or the *part-of* relation, (e.g., white-blood-cell count is *part-of* hematology lab results).

Classes representing concepts in the domain have a temporal specification of static, time-stamped, or interval. Instances of the two latter classes have corresponding instance attributes of either *time-stamp*, or *interval*, the latter of which is a pair of time points. Thus, the planning entity *medication* has instances that have the attributes *start time* and *stop time*, which denote the beginning and end of drug-administration episodes.

Expressions and Data Manipulations: In our new architecture, we take a pragmatic approach to specifying the global data model associated with a given problem solver. The global data model consists of a grammar for datatype declaration, data access, and data manipulation, with an interpreter that implements the semantics of that grammar. The grammar provides the primitive datatypes, basic terminals and nonterminals, and rules for constructing legal expressions for data access, data manipulation, and more complex datatypes. The grammar thus constitutes a modeling language in terms of which the concepts and the primitive operations in the domain can be defined.

For example, the grammar allows expressions such as “FIRST value of white-blood-cell count DURING LAST medication INTERVAL,” where the capitalized key words have meaning within the grammar for the particular data model, and the lower-case symbols are terms that either must be predefined in the method specification (e.g., *value*), or must be defined by the knowledge engineer for the domain (e.g., *white-blood-cell count* and *medication*). The grammar and its interpreter allow the semantics of the expression to be

defined in terms of the data model, rather than in terms of ad hoc symbol-level constructs. For example, one way in which the interval associated with a planning entity could be represented would be as an *n*-tuple in a relational table that stores the start and stop times and the granularities associated with those time measurements; the data model, however, shields the knowledge engineer and the domain expert from such symbol-level concerns.

The grammar for the global data model also defines the legal data manipulations. Entities that are associated with time intervals may have instances that are manipulated with temporal operators that start a new interval (creating an instance of the entity with a certain start time), that stop an interval (setting the stop time to a particular time point, thus clipping the interval), that delete an interval (removing an instance of the entity from the system's active memory), and that alter the attribute of an interval-associated entity (setting some attribute of the instance to a particular value).

3.2 Control Model

The control model of this architecture relies on a model of agenda-based execution (Firby, 1989). The run-time model of the problem solver consists of an agenda to which problem-solving tasks can be posted. The tasks can have priorities and they can be linked by a partial ordering. An agenda interpreter cyclically determines the priorities of the tasks subject to the ordering constraints. The methods or mechanisms associated with the selected task in turn are selected for execution and linked to the invocation task. Method- or mechanism-specific interpreters execute the methods or mechanisms, and possibly post more tasks to the agenda.

As the tasks and methods are executed, they incrementally write their output to a global working memory. It is not necessary for the task or method to be completed before other methods are triggered by changes in the working memory. Thus, the input and output specifications of the tasks and methods are data-flow relationships.

This agenda-based execution model allows great flexibility in specifying control information. The control information can be implicit procedural control hidden in the code of the method or mechanism interpreter, or it can be expressed explicitly in a control language that orders the subtasks in a method's subtask decomposition. Alternatively, the triggering conditions in methods or mechanisms may cause data-driven execution. In specifying the control structure in skeletal-plan refinement, we shall use all three types of control schemes.

We have developed languages for specifying both domain-dependent and domain-independent control knowledge. One example of domain-dependent control specification is the transition diagram shown in Figure 3. It specifies the chemotherapy sequence for a particular cancer clinical-trial protocol. In PROTÉGÉ-II, this control information is the input knowledge to a planning-entity decomposition mechanism that is invoked when the system attempts to instantiate the appropriate chemotherapy at a point in time. Thus, the role of the domain-specific control knowledge is defined by the mechanisms that specifically use that kind of knowledge. Our current domain-independent control language allows the specification of a partial ordering among a set of subtasks.

(Insert Figure 3 here)

3.3 Subtask and Method Specification

We see the ESPR method as being decomposed into three subtasks: proposing plan actions based on a high-level plan, identifying problems, and modifying the plan action in light of the identified problems. We shall use these three subtasks and their methods to illustrate how we specify the tasks, the methods for solving them, and the domain and control knowledge that must be acquired to generate both the knowledge-acquisition system and the problem solver. We shall assume that these three subtasks and the methods for solving them have already been entered into our task and method libraries. Furthermore, we shall assume that all the small-grained subtasks, methods, and mechanisms for configuring

these methods are in the built-in libraries. In the Section 4, we shall show how the ESPR method for the protocol-based execution planning task can be assembled from these subtasks and methods.

All three subtasks have the case data and a time point as inputs. The plan-proposal task, in addition, takes as an input a planning-entity class. The task then produces a set of planning-entity instances that have been instantiated from the input planning-entity class and from other planning-entity classes that make the given planning entity executable. The problem-identification task takes case data and a set of problems to identify as input, and generates a set of intervals representing the problems that are detected in the case data. The plan-modification task uses problems that have been identified and a set of planning-entity instances as input, and generates a new set of planning-entity instances.

In protocol-based execution planning, we use three methods—*instantiate-and-decompose*, *generalize-from-data*, and *situation-directed-revision*—to solve the plan-proposal, problem-identification, and plan-modification subtasks, respectively. These three methods use knowledge of the planning entities, the problem-abstraction mechanisms, and a set of situation-based mappings from case data to plan-modification subtasks. These three types of knowledge together constitute our model of a protocol.

The *instantiate-and-decompose* method for proposing an execution plan has as its inputs case data, a time point, and a top-level planning entity. Its subtasks are to instantiate the given top-level planning entity and to use one of the decomposition mechanisms to find those component parts of the planning entity that must be instantiated recursively at the given time point.

The *generalize-from-data* method for identifying problems uses case data and a control structure for identifying the problems from the given case data. In PROTÉGÉ's library of methods, we shall have a set of problem-detection and abstraction mechanisms.[1]

The *situation-directed-revision* method uses case data and the problems identified in the problem-identification subtask to modify the planning entities proposed in the *instantiate-and-decompose* phase. Associated with the method is a parameterized plan-modification subtask, where the parameter is one or more of the domain-dependent problems (such as renal toxicity or opportunistic infection) that are specified by the knowledge engineer or the domain expert. The control structure of this method binds problems identified in the problem-identification phase to the parameterized plan-modification subtask. For each possible combination of problems, the domain expert needs to specify the appropriate plan-modification procedure, either using the primitive data-manipulation operations in the data model or specifying another plan-proposal subtask.

4 KNOWLEDGE ACQUISTION FOR THE ESPR METHOD

The use of a global data model, a control model, and libraries of domain-independent tasks and methods as the basis for expert-system development has implications for building a knowledge-acquisition system.

The knowledge-acquisition system must have facilities that allow the entry of the data query and manipulation expression allowed in the grammar of the global data model. This facility is needed everywhere that the specification of an expression is required and, thus, must be implemented as an editor that can be embedded in other editors.

Moreover, the knowledge engineer will be defining the domain concepts used in the methods by *subclassing* the input and output classes of tasks and methods or by *composing* existing classes into composite classes. Subclassing an entity class involves specifying values in some class attributes and defining additional domain-specific attributes. The

[1] See (Shahar, 1992) for definitions of the temporal-abstraction mechanisms and the requirements for instantiating these mechanisms.

PROTÉGÉ-II system uses the input and output class declarations of the prebuilt tasks and methods, and generates forms that allow the knowledge engineer to define specializations of those classes. These domain-specific subclasses defined by the knowledge engineer in turn determine the forms in the knowledge editor that PROTÉGÉ-II creates for the domain experts.

Composing a class means declaring a new class and specifying that the attributes of that class take existing class objects (or their subclasses) as values. In the example to be discussed, we shall declare a *protocol* as a composite class whose attributes are planning entities, problems, and plan-modification actions. The problem-solving methods that act on composite classes, such as the *protocol*, access the attributes of the composite classes through predefined class access functions provided by the data model.

The control model in PROTÉGÉ-II includes languages for specifying both domain-independent ordering of subtasks and domain-dependent flow of control. Both languages must have associated editors through which the control information can be entered. As the domain-dependent control-flow editor will be used by application specialists, the possible datatypes used by the editor must be couched in domain terms (e.g., sequencing of *medications* to administer). Thus, the control-flow editor must be instantiated for a particular domain.

Finally, the knowledge-acquisition system must have the facility to allow a knowledge engineer to associate methods from the method library of PROTÉGÉ-II with the tasks and subtasks selected for the problem-solver. The knowledge-acquisition system must index the methods and tasks to help the knowledge engineer select appropriate methods for the tasks (Puerta et al., 1992).

Using these facilities provided by PROTÉGÉ-II, the knowledge engineer first assembles and configures the problem-solving method for the task (Section 4.1). The input and out-

put specifications of these configured methods define the knowledge roles for which domain-specific datatypes must be defined (Section 4.2). Using the configured methods and mechanisms and the domain-specific knowledge roles defined by the knowledge engineer, PROTÉGÉ-II generates a knowledge editor that can be used by application specialist to enter the necessary domain knowledge (Section 4.3).

4.1 Assembling the ESPR Method

To assemble the ESPR method, the knowledge engineer must make a number of design decisions. As we have seen, there are numerous data-flow relationships among the subtasks. These data-flow relationships only partially constrain the possible control flow. For example, it is possible that, before the propose-plan subtask has completed execution, the problem-identification subtask can be invoked with some problems to check. The knowledge engineer can fix the order of the invocation of the subtasks by placing a partial ordering on the subtasks. In Figure 4, we force the plan-modification subtask to execute only after the initial plan-proposal subtask has completed, but we place no such constraints between the plan-proposal and problem-identification subtasks.[2]

The output specifications of the configured method can be derived easily from the outputs of the method's constituent subtasks. The knowledge engineer should select a subset from the union of the outputs of the constituent subtasks. In the ESPR method that we are configuring, we may choose the problems the system has identified, as well as the planning-entity instances, as the outputs of the method.

(Insert Figure 4 here)

[2] A more general architecture will have metalevel control methods (strategies) that determine the order of the execution of the subtasks dynamically.

The inputs to the top-level constructed method are more difficult to define. It is not sufficient to take the union of the inputs of the constituent subtasks. Recall that the input data objects of a method include not only the case data, but also the knowledge used in solving the problem. For PROTÉGÉ-II to generate a knowledge editor for a clinical-trial management task, it must have a coherent model of the protocol in the domain. Thus, the inputs to the ESPR method that we are constructing cannot be specified completely until the knowledge requirements for methods that solve the subtasks are known. It is from these knowledge requirements that we construct a model of the protocol and any other domain knowledge required for solving the protocol-based execution-planning task.

To develop a model of the protocol, we must abstract the knowledge used in all the methods and mechanisms that may be used to solve the subtasks in the task decomposition of the assembled ESPR method. The knowledge engineer must use the class-composition capability of our data model to declare a protocol as a class object that contains (1) a set of planning entities and their associated decomposition mechanisms, (2) a set of problems and their associated problem-identification mechanisms, and (3) a set of plan-modification mechanisms.

4.2 Defining the Domain Task Model

Once the overall problem-solving method has been selected or assembled, knowledge acquisition is driven by the requirements of operationalizing the method in a particular domain. The inputs of the methods must be specialized to terms meaningful in the domain. For illustration purposes, we shall use clinical trials of drugs for treating HIV-positive patients as the application domain.

Given the input requirements of the ESPR method assembled in Section 4.1, PROTÉGÉ-II will ask the knowledge engineer to define a set of case-data objects representing concepts in the domain (such as white-blood-cell counts or bilirubin), a *part-of* hierarchy of planning-entity classes, a set of possible problem abstractions, and a set of revision mech-

anisms that map problems and data to plan-modification actions. For each subclass that the PROTÉGÉ-II developer defines, she should define the class and instance attributes associated with that subclass. For each attribute, the system needs to know the legal values, as well as the time when the values are determined (knowledge-editor time or run time). We shall briefly illustrate some of the planning entities and problem abstractions that a knowledge engineer may specify for the HIV domain.

Figure 5 shows one planning-entity part-of hierarchy for the HIV domain. This specification of planning entities indicates that protocols in this domain have three levels of abstractions for therapeutic actions: (1) protocol administration as a whole, (2) regimens for alternative types of treatment, and (3) medications making up a regimen. This structure will determine the types of domain knowledge that the PROTÉGÉ-II-generated knowledge editor will request from the domain expert.

(Insert Figure 5 here)

For each class in the planning-entity composition hierarchy, the PROTÉGÉ-II knowledge engineer indicates what subtasks for finding instance-attribute values should be set up in the instantiation process. The semantics of these attributes will be defined by the mechanisms that the knowledge engineer associates with these subtasks. We have identified a small number of mechanisms for specifying relationships among attributes of planning entities. One such mechanism allows specification of definitional relationships among attributes of entities. For example, the current cumulative dose of the drug can be defined as the sum of a previous cumulative dose and the product of the current dose and frequency. Other relationships among attributes will be defined by the domain specialists using the mechanisms made available from the method library.

Figure 6 shows part of a problem hierarchy that can be defined for the HIV domain.[3] The problem list defined by the PROTÉGÉ developer forms the menu of problem classes that

the knowledge-editor user can use to define protocol-specific problems for which therapy actions may be altered.

(Insert Figure 6 here)

4.3 Knowledge Acquisition Via Knowledge Editors Generated by PROTÉGÉ-II

Given planning-entity and problem class hierarchies such as those shown in the previous section, PROTÉGÉ-II generates a knowledge editor custom-tailored for the domain. Instead of using generic terms such as*planning entity*, the knowledge editor uses domain-specific terms such as *medication.*

For each domain-specific planning-entity class, the user of the knowledge editor defines subclasses that are used in that protocol. The knowledge editor has forms through which the attribute values of these planning entities can be specified. For planning entities that have part-of children, the user is required to specify the decomposition method by selecting from the library of mechanisms the appropriate decomposition mechanism, and to specify the necessary data for that mechanism.

Figure 7 shows how the planning entities of a protocol (protocol CCTG-522) can be defined as subclasses of the planning entities defined at the PROTÉGÉ level. A decomposition mechanism associated with a planning entity may instantiate some of the planning entities at lower level, or a plan-modification mechanism may cause a planning entity to be instantiated. Each of the mechanisms has its own editor. For example, if the domain expert wants to specify a sequence of treatment actions similar to the one in Figure 3, the

---

[3] The problems shown in Figure 6 are abstractions about patient *states*. We have also developed mechanisms for making *gradient* and *rate* abstractions that characterize direction and rate of change of parameter data (Shahar et al. 1992).

knowledge editor brings up the graphical editor for specifying such a procedure (Puerta et al. 1992).

(Insert Figure 7 here)

The knowledge-editor user selects from the problem class hierarchy to specify the problems that are relevant to a particular protocol (Figure 8). The user should be able to define protocol-specific concepts (e.g., a *dose-reducing-toxicity* in CCTG-522). There shall be a set of problem-abstraction mechanisms with which the domain expert can specify how a problem is to be derived from input data or lower-level abstractions (Shahar et al. 1992). The knowledge editor will check for completeness by making sure that an abstraction mechanism has been instantiated for every problem mentioned in a protocol.

(Insert Figure 8 here)

DISCUSSION

The first PROTÉGÉ system built in our laboratory assumed that the application task for which it was designed could be viewed in terms of the ESPR model. The tool provided a language for expressing domain concepts in terms that a special-purpose inference engine (e-ONCOCIN) could use. This assumption of a fixed method and the consequent fixed knowledge roles allowed a knowledge engineer working with PROTÉGÉ to generate graphical knowledge editors that were tailored to the application area. These editors could be used by domain specialists to enter the detailed content knowledge required by the problem solver.

The assumption of a fixed problem-solving method, however, made PROTÉGÉ brittle in two respects. First, the system by definition cannot be used for tasks and methods for which it is not designed. Even for a given task, there may be several alternative solution methods, each of which is more appropriate in certain situations than in others. Second, a

monolithic method cannot anticipate all possible variations in the ways that knowledge is applied in a domain. The method is applicable in only domains where the knowledge requirements fit the method exactly, down to the most detailed level of abstraction. For those aspects of the knowledge that depend on domain-dependent considerations (e.g., a procedure for calculating the current dose of a drug), the skeletal-planning model in PROTÉGÉ defines no explicit knowledge roles that users can simply fill in.

In our current work, we deal with the problem of brittleness in three ways. First, we generalize the PROTÉGÉ framework so that, instead of using one fixed problem-solving method, our system has libraries of tasks and methods for solving those tasks. Configuring or assembling the problem-solving method becomes the first step in the knowledge engineer's interaction with PROTÉGÉ-II. This ability to tailor problem solvers for alternative tasks extends the range of the tasks for which PROTÉGÉ-II can be used, and allows the knowledge engineer to associate multiple methods with the same task. Second, the uniform view of problem solving as using methods to accomplish tasks allows us to use the same type of knowledge-level analysis at multiple levels of granularity. For a small-grained task, such as making an abstraction from data, we can define correspondingly small-grained domain-independent data-abstraction mechanisms. As a method is decomposed into subtasks that have their own solution methods, selecting a top-level method does not have to fix the methods for the subtasks. This capability to define problem-solving methods at multiple levels of granularity allows great flexibility in custom-tailoring the problem-solving behavior for a particular task. Third, we emphasize the use of a *data model* and of a *control model* that provide general-purpose languages for configuring and assembling problem-solving methods. These models are tools that a knowledge engineer uses for custom-tailoring the knowledge editors and the problem-solvers generated by PROTÉGÉ-II.

We allow for the specification of domain-dependent control knowledge by defining languages (such as the flowchart language used in Figure 3) for expressing such knowledge and by defining specialized mechanisms that interpret them. These mechanisms are invoked within specific methods (e.g., the flowchart in Figure 3 is interpreted by a decomposition mechanism that is invoked in only the *instantiate-and-decompose* method). Thus, we allow the specification of domain-dependent control knowledge, but only in constrained contexts. Such constrained use of domain-dependent control knowledge gives us the necessary flexibility while retaining the advantages of role-limiting methods.

We recognize that developing a knowledge-based system is a complex task requiring both the modeling expertise of a knowledge engineer and the domain expertise that only an application specialist can provide. We address both requirements, as in the first version of PROTÉGÉ, by dividing the knowledge-acquisition process into two phases. In the first phase, the knowledge engineer uses the modeling tools of PROTÉGÉ-II, such as the libraries of tasks and methods and data-model and control-model languages, to configure a problem-solving method for the task and the domain. Based on the knowledge requirements of the configured method, PROTÉGÉ-II generates a domain-specific knowledge editor that the domain specialist can use to enter the necessary domain knowledge.

Our group's project to seek ways of extending PROTÉGÉ's method-oriented architecture parallels work by a number of other researchers to develop ways to adapt problem-solving methods to the needs of particular domain tasks. For example, researchers at the Free University in Brussels have developed a componential framework (Steele, 1990) and have implemented a prototype testbed (Vanwelkenhuysen, 1990) that bear similarities to the PROTÉGÉ-II architecture. The componential framework requires the system developer to analyze a task in terms of input and output, available domain knowledge, case model, and pragmatic constraints—such as incompleteness of the data—that may be present in the domain. A problem-solving method in the componential framework may have domain

models that decompose a task into subtasks or it may solve a task directly. The implementation uses a frame system that allows representation of each component of the framework as an object that can be tailored dynamically to the problem at hand.

PROTÉGÉ-II differs from the componential framework more in degree than in kind. In the componential framework, domain knowledge is part of the task characterization, and the method contains the procedural control structure. In PROTÉGÉ-II, on the other hand, both the knowledge and the control structure are associated with the problem-solving method. Although the componential framework is consistent with a two-phased approach to knowledge acquisition, such as PROTÉGÉ methodology, Steels (1990) sees the development of knowledge-based systems as primarily the responsibility of the knowledge engineer. Our emphasis on the division of labor between knowledge engineers and domain experts leads us to develo a set of data and control models that knowledge engineers can use to configure and assemble methods, and to explore techniques for generating domain-specific knowledge-acquisition systems from such configured methods.

Faced with brittleness problems in the Generic Task architecture (Chandrasekaran, 1986), investigators at Ohio State University are developing to an approach in which generic tasks are implemented in SOAR (Johnson et al., 1990). Each generic task is represented within a SOAR problem space (Laird et al., 1986). As such, the inputs to the generic task are modeled as the initial state of the problem space; the desired output of the generic task is modeled as the goal state; and the problem-solving method is modeled using operators available within the problem space. The Ohio State group hopes that SOAR's universal weak method will provide a common foundation for all generic tasks, and that SOAR's universal subgoaling behavior will facilitate the integration of multiple generic tasks that each may contribute to problem solving at different levels of granularity. A significant advantage of the SOAR approach is that the weak methods in SOAR provide a default problem solver when specific domain knowledge is not available. It is not clear, however,

how the SOAR representation itself can make explicit the roles in which knowledge is used in problem solving to facilitate the generation of knowledge-acquisition tools like those created by PROTÉGÉ.

McDermott's group at Digital Equipment Corporation shares our view that problem-solving methods can be composed from more primitive building blocks called mechanisms (Klinker et al., 1991), 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 help a developer to select mechanisms from a library, and will combine these mechanisms to construct problem-solving methods accommodating the requirements of particular application tasks. The output of Spark is a knowledge-acquisition tool (called Burn) that will allow users to enter specific domain knowledge.

Despite strong similarity, PROTÉGÉ and Spark are being designed for different communities of users. Our research group assumes that knowledge engineers will be the principal users of PROTÉGÉ and that these engineers will want to configure new problem-solving methods explicitly. The developers of Spark, however, anticipate that the primary users of their tool will be nonprogrammers; thus, they are building Spark to select and modify configurations of mechanisms programmatically based on features of the application task (Klinker et al., 1991). Spark itself does not address the problem of how to guide knowledge engineers in their assembly of different mechanism configurations; in contrast, providing such guidance is a major concern for the new version of PROTÉGÉ. Our group is experimenting with a number of visual languages that may help knowledge engineers to construct complex datatypes in the method-configuration process and to view complex relationships among mechanisms (Puerta et al., 1992).

It is encouraging that several research groups are converging on similar methodologies for developing knowledge-based systems. In all these projects, considerable attention is given to identifying what problem-solving abstractions are appropriate and useful and how these

small-grained abstractions might be combined into useful problem-solving methods. Although there are important differences in the manner in which the various architectures allow users to assemble and instantiate problem-solving methods, a primary challenge faced by all groups is describing, at the knowledge level, appropriate tasks and methods. This common framework raises the possibility that these descriptions of problem-solving behavior can be shared across the research groups.

This paper reports on work of an ongoing project. As we implement the problem-solving abstractions for the method of ESPR and test similar ideas in other application areas, we will refine our languages for describing data, domain knowledge, and control information. The framework described in this paper will give us a basis for developing custom-tailored knowledge-acquisition tools, and flexible and extensible problem solvers.

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.

John Egar and Josef Schreiner have contributed greatly to our new design for PROTÉGÉ and have provided valuable comments on previous drafts of the paper. Lyn Dupré edited the manuscript and saved the first author from many grammatical and stylistic embarrassments.

REFERENCES

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

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

Firby, R.J. (1989). *Adaptive Execution in Complex Dynamic Worlds*. Ph.D. Dissertation, Department of Computer Science, Yale University, New Haven, CT.

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

Johnson, T.R., Smith, J.W., Chandrasekaran, B. (1990). Task-specific architectures for flexible systems. Technical Report. Department of Computer Science, Ohio State University, Columbus, OH.

Klinker, G., Bhola, C., Dallemagne, G., Marques, D., McDermott, J. (1991). Usable and reusable programming constructs. *Knowledge Acquisition*, **3**, 117–135.

Laird, J., Rosenbloom, P., Newell, A. (1986). *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies*. Boston: Kluwer Academic.

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

Musen, M.A., Fagan, L.M., Combs, D.M., 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. (1989). *Automated Generation of Model-Based Knowledge-Acquisition Tools*. London: Pitman.

Puerta, A., Egar, J., Tu, S., Musen, M. (1992). A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. *Knowledge Acquisition*, **4**, xxx–xxx.

Shahar, Y., Tu, S.W., Musen, M.A. (1992). Knowledge acquisition for temporal abstraction mechanisms.*Knowledge Acquisition*, **4**, xxx–xxx.

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

Tu, S.W., Kahn, M.G., Musen, M.A., Ferguson, J. C., Shortliffe, E.H., Fagan, L.M. (1989). Episodic monitoring of time-oriented data for heuristic skeletal-plan refinement. *Communications of the ACM*, **32,** 1439–1438.

Vanwelkenhuysen, J., 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.