Automated Generation of Adaptable Knowledge-Acquisition Tools with Mecano

Angel R. Puerta, John W. Egar, and Mark A. Musen

Medical Computer Science Group
Knowledge Systems Laboratory
Departments of Medicine and Computer Science
Stanford University
Stanford, CA, 94305-5479
puerta@camis.stanford.edu

ABSTRACT

Method-oriented knowledge-acquisition tools are based on a model, or method, of problem solving and can acquire knowledge for the class of tasks that can be solved with that particular problem-solving method. The capture of knowledge takes place in knowledge editors. These editors are typically based on the individual tool's domain-independent method; they fail to reflect task- and domain-specific characteristics and have no ability to adapt to user requirements. Mecano is a user-interface management system that generates automatically adaptable knowledge editors for the PROTÉGÉ-II knowledge-acquisition shell. Mecano allows knowledge engineers to specify the components of a knowledge editor independently of any underlying problem-solving method. It also provides facilities for constraining the operations allowed on the components, for selecting interaction styles for each component, and for linking components to coordinate their simultaneous display. Knowledge editors generated by Mecano take into account the needs and requirements of given tasks, domains, and users, and guide the users through the knowledge-editing process by providing visual cues and by limiting the permissible editing operations to those relevant in the domain of interest.

KEYWORDS: Adaptive systems, user-interface management systems, knowledge editors, knowledge acquisition.

INTRODUCTION

One of the most challenging tasks in the development of knowledge-acquisition tools is the specification of a user interface that allows domain experts to enter knowledge into a knowledge base. This interface usually takes the form of a *knowledge editor.* A knowledge editor permits a user to view, modify and augment the contents of a knowledge base. Existing knowledge editors have varied interaction styles [9]. One commonality in most of these tools, however, is the *manual* specification by the tool developers of the knowledge editor. Examples of this type of interface specification are SALT [7], a tool for acquiring knowledge on constraint-satisfaction tasks, such as scheduling, and ROGET [2] a tool for problem-diagnosis tasks. One exception to the manual definition of knowledge editors is PROTÉGÉ [8]. This knowledge-acquisition tool operates at the *metalevel* and generates *automatically* a knowledge editor for a given task.

All three tools mentioned belong to a class of architectures called *method oriented* because they assume a single model, or method, of problem solving for the tasks for which knowledge is to be acquired. Thus, ROGET assumes a variation of heuristic classification [3], SALT a propose-and-revise problem-solving strategy [7], and PROTÉGÉ a form of the model of skeletal-plan refinement [8]. Presupposing a particular model of problem solving has two major limitations. First, the usefulness of method-oriented tools is limited to the class of tasks that can be solved with the given problem-solving strategy. Second, since these problem-solving methods are domain-independent, the knowledge editors developed—or generated, as in the case of PROTÉGÉ—are based on the characteristics of the problem-solving methods and do not account for the needs and requirements of specific tasks, domains, or users. The result is that the interfaces of the knowledge editors follow the computational requirements of the knowledge-acquisition tools, rather than the cognitive requirements of the users of the knowledge editors.

To eliminate the restrictions to applicability of method-oriented architectures, we are developing in our laboratory PROTÉGÉ-II [10,11], a knowledge-acquisition *shell*, operating at the metalevel, that does not presuppose any particular problem-solving method and that allows

knowledge engineers to build knowledge editors for different methods. Since it is not possible to know beforehand which method will be selected, the automatic generation of knowledge editors in this shell cannot be based on the characteristics of a particular method, as it was in the original PROTÉGÉ system. In this paper, we present *Mecano,* a user-interface management system for PROTÉGÉ-II that provides a common framework for the automatic generation of knowledge editors under multiple problem-solving methods. Mecano also deals with the shortcomings of not taking into account task, domain, and user requirements by generating knowledge editors that are *adaptable.* Using the facilities of Mecano, knowledge engineers can adapt the knowledge editors produced by Mecano to the characteristics of specific tasks, domains, and users.

Although other researchers of knowledge-acquisition tools are examining means to develop multiple-method architectures [6,12], they have generally disregarded the problem of developing knowledge editors that are suited for individual tasks and domains. As a consequence, even when a knowledge editor can be built for a given task, its usefulness may be highly constrained because the interface of the knowledge editor does not allow users to enter knowledge in a way that is natural in the domain of interest. In Spark [6], a metatool that assists nonprogrammers in automating tasks, the interface for knowledge acquisition is general purpose and fails to adapt to single tasks or domains. In DOTS [4], a metatool that is not method-oriented, it is possible to build task- and domain-specific knowledge editors. The process for each new knowledge editor, however, requires the developer to write the complete specification of the interface using a toolkit layer that resides on top of the host window system. The aim of Mecano, on the other hand, is to automate, as much as possible, the construction of a new knowledge editor and to provide functionality to fit such an editor to the requirements of the given task, domain, or user.

THE MECANO APPROACH

Mecano views the problem of generating a knowledge editor as one of writing a specification for an interface. Thus, facilities must be provided by the system to define the interface components, to determine the operations allowed on the components, and to establish the relationships among components. We can best discuss how Mecano covers the required functionality by examining its role within the context of the PROTÉGÉ-II knowledge-acquisition shell. This environment is used by knowledge engineers to build new knowledge editors for specific tasks and domains. The editors are in turn used by domain experts to enter knowledge into knowledge bases, or to edit knowledge already entered. The creation and use of a knowledge editor is accomplished through a process that involves the four phases shown in Figure 1. Although we shall describe all phases, we shall concentrate on the bottom three in Figure 1, since those are the ones affected by the actions of Mecano.

The process of building a new knowledge editor begins with the *method-selection phase.* During this phase, the knowledge engineer searches through a library of methods for a problem-solving strategy that best fits the task of interest [11]. Normally, a certain amount of custom tailoring of existing library methods is necessary; in some instances, if the search does not yield any suitable candidate methods, the knowledge engineer may be required to define a new problem-solving method using PROTÉGÉ-II. Ultimately, at the end of this phase, a single, domain-independent method will be specified and its characteristics will determine how the next phase is conducted. The selected method will impose requirements on what knowledge is necessary to solve a particular task, how the task is solved, and what is the role that each piece of knowledge will play in the solution of the task.

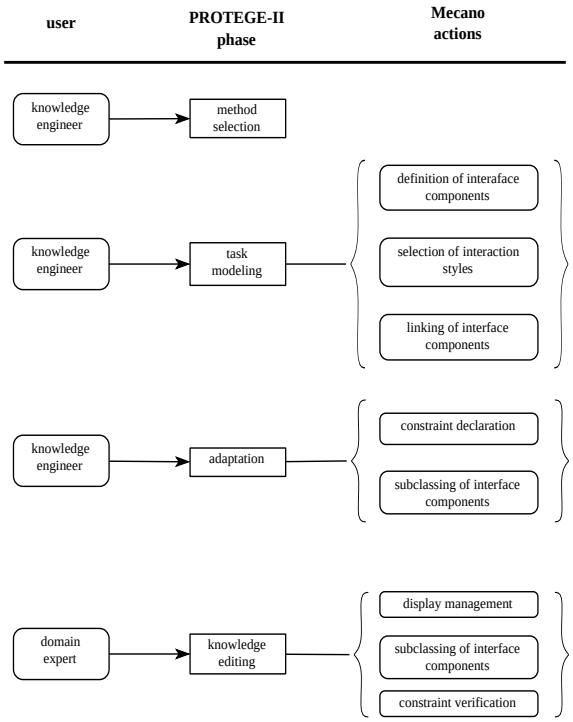

Figure 1: The four phases for the definition of new knowledge editors in PROTÉGÉ-II. Mecano supports functionality for the definition of interface components for the knowledge editor, for the selection of interaction styles for each component, for the declaration of constraints to limit the types of operations allowed on such components, and for the specification of dependency links among components to manage simultaneous display of dependent components.

The knowledge requirements imposed by a method create a framework for the definition of domain-specific concepts in terms of the problem-solving method. Thus, during the *task-modeling phase,* the knowledge engineer examines the given task in the context of the selected method and identifies the relevant domain concepts and the role that they play in solving the task problem. The result is a model of a task that is used for two purposes: (1) generating a knowledge editor to enter the required knowledge into a knowledge base, and (2) supplying an inference engine with a problem-solving model with which to reason about the knowledge present in the knowledge base [10]. It is during this phase that Mecano starts gathering the specifications needed to generate a knowledge editor. Users can stipulate which domain concepts will be represented in the knowledge editor as interface components. A knowledge-editor interface component, for our purposes, is any interaction element (e.g., a form blank, or a graphical element in a graphical editor) that can be employed to capture the needed knowledge from the domain expert. In addition, users can choose the interaction style for each defined interface component. The choice is normally driven by the type of knowledge to be acquired (e.g., procedural, or factual). Furthermore, the user can link together defined interface components, so that linked components will be displayed simultaneously when any of the linked components is selected.

Once the user completes the model of the task, Mecano enters an *adaptation phase,* during which it generates a *preliminary* knowledge editor with all the interface components declared during task modeling. Normally, the interface components fall into two categories: graphical elements in a palette, or blanks in forms. The graphical elements are used to draw diagrams in a graphical editor; the form blanks are used to enter facts. The generated editor is considered preliminary at this point because of its adaptable nature. The knowledge engineer can modify the editor such that task- and domain-specific characteristics are taken into account. The adaptation takes place by means of constraint declarations that affect how the interface components are manipulated by the domain expert, and by subclassing of components, which permits changes to the appearance of the components in the editor. Previous method-oriented architectures could not account for the characteristics of individual tasks and domains because there was a direct, unchangeable path between the task model and the knowledge editor. Given that the task models are derived from domain-independent methods that apply to a broad class of tasks, task- and domain-specific considerations were absent from the knowledge-editor construction process. We added an adaptation phase, so that Mecano can permit the definition of task models from domain-independent methods while allowing for the incorporation of single-task and single-domain features.

After the user adapts the editor, Mecano manages the editing sessions with the domain expert during the *knowledge-editing phase.* A *constraint verifier* alerts the user to illegal operations with the interface components, and a *structure manager* coordinates the display of linked components. In addition, the system keeps track of any further subclassing of knowledge-editor interface components that may be needed in this phase in order to maintain proper relationships between the new subclasses and the task model.

In the next three sections, we shall illustrate the use of Mecano with an example taken from the medical domain. The overall effect of the use of the features of Mecano is the production of knowledge editors that not only serve as environments for knowledge acquisition, but also guide the domain expert through the acquisition process by providing visual cues and by limiting the number of allowed user actions to those that are valid for the task for which knowledge is being captured.

THE TASK-MODELING PHASE

The example that we shall discuss assumes that a knowledge engineer is using PROTÉGÉ-II to build a knowledge editor to capture knowledge about treatment plans (called *protocols*) for cancer patients. During the method-selection phase, the knowledge engineer selected *skeletal-plan refinement* [5] as the problem-solving method that will be applied during task modeling. This particular method solves problems by creating solution plans. It starts with an abstract (skeletal) plan, then refines this plan by decomposing it into constituent plans recursively until a final, fully detailed solution plan is completed. The strategy is suitable for cancer protocols because a protocol can be viewed as a fully detailed plan for the treatment of patients.

In skeletal-plan refinement, models of tasks are built around three concepts: planning entities, input data, and actions. *Planning entities* are problem-solution components (plan components) that can be decomposed into other planning entities thereby forming a hierarchy. *Input data* are extracted from the associated environment. In our case, data will take the form of, for example, laboratory-test results. *Actions* are procedures that can modify planning entities. For example, drug-dosage attenuation is an action that must be modeled in our domain. The methods available in the library of PROTÉGÉ-II include in their definitions an interface specification for the modeling of tasks. Figure 2[1] shows part of the task-modeling interface for skeletal-plan

1. Legibility of screen snapshots is limited by technical limitations. A high-resolution linotropic printer will be available for the final draft of this paper.

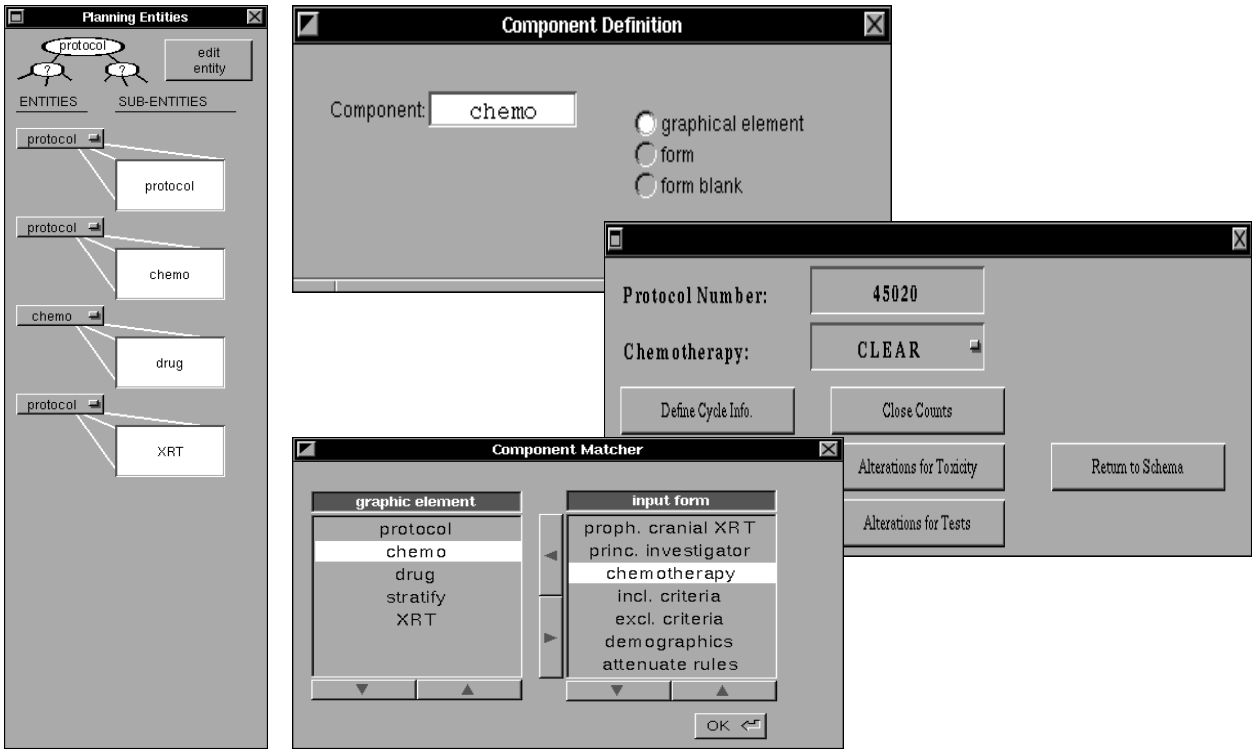

Figure 2: The task-modeling phase. The knowledge engineer defines which domain concepts will be part of the knowledge editor and what interaction style will be used to display them. In addition, the knowledge engineer can link two components, such as "chemo" with the "chemotherapy" form (shown at right), to cause their simultaneous display during the knowledge-editing phase.

refinement. In the figure, the knowledge engineer is declaring the planning-entity hierarchy for the domain of oncology. The hierarchy determines that plans in this domain are made up of protocols. The protocols can be decomposed into chemotherapies (administration of drugs to patients) and radiotherapies (administration of X-ray treatments to patients), labeled "chemo" and "xrt" in Figure 2, respectively. Chemotherapies can be further decomposed into drugs. Using separate parts of the task-modeling interface, the knowledge engineer defines all remaining relevant domain concepts (input data, actions) under the terms of the domain-independent skeletal-plan refinement method.

In addition to the task-modeling interface, Figure 2 also depicts two Mecano windows that allow the definition of components of the knowledge-editor interface: selection of interaction styles, and linking of interface components. The knowledge engineer is designating "chemo" as one of the components and declaring that this component must be represented in the palette of graphical elements that Mecano generates. Currently, only two interaction styles are available: graphical-based and form-based. The knowledge engineer chose graphical-based interaction for the component "chemo" because protocols involve procedural knowledge and are usually represented by physicians as flowcharts. Thus, the intention of the knowledge engineer is to enable expert physicians to employ the palette of graphical components to draw flowcharts in a graphical editor during the knowledge-editing phase, making the knowledge-acquisition process similar to their usual protocol-authoring activities.

Although the concept of chemotherapy is declared by the user, in this example, as a graphical-style component, certain attributes related to the concept—such as duration, number of drug-administration subcycles, and duration of each subcycle—cannot be expressed efficiently in a graphical style. Therefore, Mecano features a formal language, called FormIKA [1], for the declaration of forms as interface components. FormIKA permits the specification of the appearance and layout of forms, and the definition of display dependencies among forms, which in effect creates a tree of forms that can be navigated by the user. During task modeling, the knowledge engineer can also stipulate display dependencies between graphical-style components and forms, or between graphical-style components and individual blanks in forms. In our example, the user is linking the component "chemo" with the form "chemotherapy" (see Figure 2). This form is the

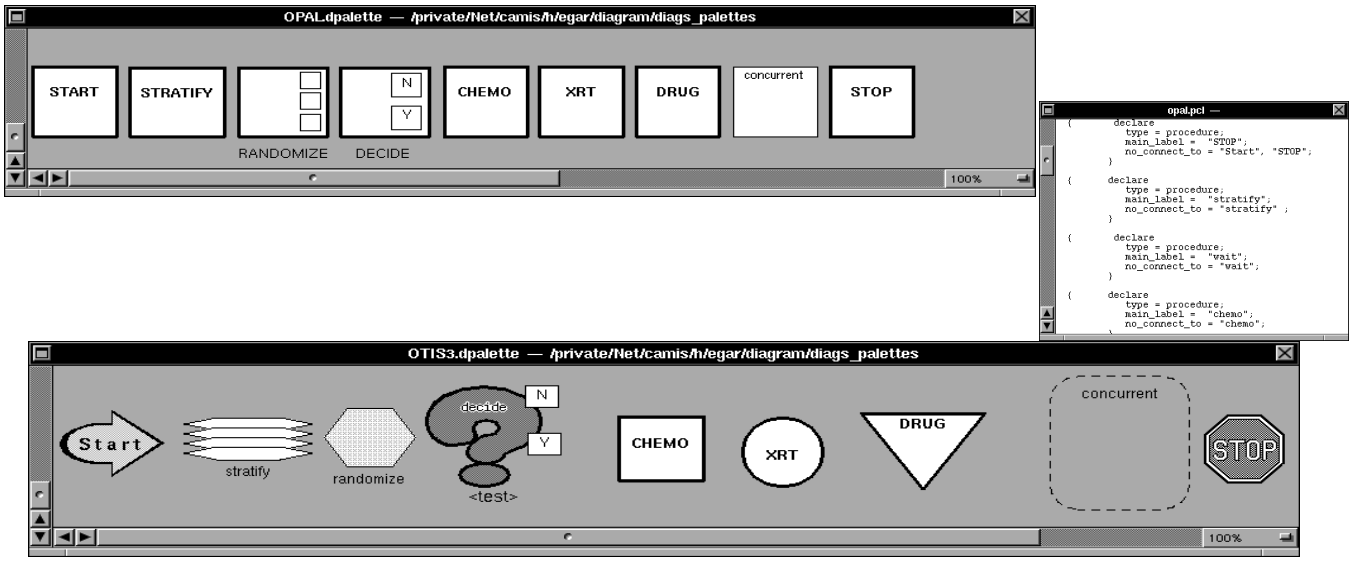

Figure 3: The adaptation phase. The palette at the top is part of the Mecano-generated preliminary knowledge editor. In the bottom palette, the knowledge engineer has defined subclasses of the components in the top palette to make these more expressive. The text window shows part of the declaration of connectivity constraints affecting the components in the bottom palette.

top node in a tree of forms that the user can be access by clicking on the buttons displayed on the form. During the knowledge-editing phase, the link determines that, every time that a new chemotherapy is defined by the domain expert, the linked form is displayed to the expert to collect all the required information on the new chemotherapy. This feature relieves the domain expert from the burden of having to know which form, or which piece of information, must be provided every time a new protocol, chemotherapy, or radiotherapy is declared. By linking the interface components in this manner, Mecano is able to guide the domain expert through the knowledge-editing session. The expert is neither presented with a bewildering array of interface components, nor required to understand the role of each one in relationship to all others. Instead, Mecano generates knowledge editors that display a few components that serve as a gateway to the rest of the interface components in a logical and orderly manner.

THE ADAPTATION PHASE

The interface specifications gathered during the task-modeling phase are used by Mecano to generate a *preliminary knowledge editor.* This editor, although fully functional, yet is adaptable; during this phase, it will be modified by the knowledge engineer to fit more closely the needs and requirements of the given task and of the target domain-expert users. The adaptability of the preliminary editor is permitted by the object-oriented approach that Mecano uses to define interface components. Every component declared during the task-modeling phase is assigned a class. The process of adapting the knowledge editor then becomes one of defining subclasses of the component classes that reflect more accurately the peculiarities of individual tasks and domains, and that are tailored to accommodate the needs of the expected users of the knowledge editor. In addition, knowledge engineers can define constraints that apply to the component subclasses and that will be enforced during knowledge editing. In this manner, Mecano bridges the gap between an editor derived from a domain-independent problem-solving method and an editor built for a specific task.

The preliminary editors generated by Mecano consist of a palette of graphical elements, a set of forms, and a set of links between forms and graphical elements. Figure 3 shows the palette of components of the preliminary editor for our oncology therapy example. Complementing the components declared during task modeling (“chemo,” “xrt,” “drug,” and “stratify”) are other components that belong to a special class of *control* components that Mecano defines as primitives, and from which the user simply selects those needed in the current task. In Figure 3, the user has defined—by using a browser similar to that of Figure 2—subclasses of the palette components in the preliminary editor to change the appearance of the components and thus to make them more expressive within the context of our example problem. Furthermore, the user has declared constraints that affect the connectivity of the palette components. In this case, connections such as “start-stop” and “chemo–chemo” have been disallowed. By imposing such constraints, the knowledge engineer ensures that the operations on components that the domain expert

will be able to perform exclude those that do not apply to oncology therapy.

The stipulation of constraints for palette components is achieved through a formal language called Palette Constraint Language (PCL). In the case of forms, constraints that affect the blanks in the forms are specified in the FormIKA language. Both languages allow constraints to be defined declaratively, as opposed to requiring the knowledge engineer to write procedural attachments to the interface components. The constraints declared in FormIKA are of two types: value and visual. A *value constraint* subordinates the value displayed in a blank to the value displayed in another blank. Therefore, when the user updates the latter, the subordinate blank is updated automatically. In the case of a numeric value, the update may be calculated through a mathematical formula. On the other hand, a *visual constraint* provides visual cues to the user by activating or disabling blanks according to the actions taken by the user. To illustrate the use of FormIKA constraints, we consider the "chemotherapy" form displayed in Figure 2, which constitutes the top of a tree of forms to enter facts about a chemotherapy. Because a value constraint has been declared, once the user enters a name in the "Chemotherapy" blank, that name is propagated to all the other forms in the tree that are available by clicking on the respective buttons in the top form. Due to a visual constraint in the "Define Cycle Info" subform (not shown), once the user enters the name of a new subcycle of chemotherapy administration, a blank is highlighted prompting the user to enter the number of days that the subcycle lasts.

The combination of links between palette components and forms, constraints on palette-component connectivity, and constraints on form blanks enables the knowledge editors generated by Mecano to guide the domain expert through the editing session. The links and FormIKA constraints create a *suggested path* to follow for the domain expert by having Mecano either display a form that must be filled in or activate a blank that must be updated. The PCL constraints define a *illegal path*s away from which the domain expert is steered by displaying alert panels that advise the expert user on the illegal operation attempted.

THE KNOWLEDGE-EDITING PHASE

The role of Mecano during the knowledge-editing phase is one of management of the editing session. Mecano coordinates the display of linked interface components, verifies that PCL constraints are not violated, and controls the declaration of new components by the domain expert. The domain expert declares new editor-interface components by further subclassing the existing interface components.

Figure 4 shows a snapshot of an editing session for the cancer-therapy example. Components from the palette are dragged into the drawing area of a graphical editor, and are connected to represent the procedural knowledge inherent to an oncology protocol. In some instances, the user must subclass a palette component when it is dragged into the drawing area. For example, the domain expert has declared two subclasses of "chemo" in Figure 4: "VAM" and "POCC." The declaration of subclasses takes place in a browser similar to that shown in Figure 2 for the linking of components. Immediately after the declaration of a new subclass of "chemo," Mecano displays the "chemotherapy" form as was stipulated by the knowledge engineer. The domain expert can then proceed to enter the information required for the new chemotherapy by following the visual cues offered in the "chemotherapy" form.

When the domain expert declares a subclass of an interface component, the new subclass inherits all the constraints that affect the superclass. Thus, the subclass "VAM" inherits the constraint that disallows the "VAM–VAM" connection. Because of this constraint, it is not possible to connect "VAM" to itself. Figure 4 depicts a panel alerting the user that this particular constraint is being violated in the current protocol.

LIMITATIONS AND FUTURE WORK

Although Mecano can generate editors independently of the selected problem-solving method, the choice of interaction styles for the declared interface components is limited. Presently, a component can be only a graphical element in a palette, a form, or a blank in a form. As a result, it is difficult to specify knowledge editors for tasks, or for problem-solving methods, that require different interaction styles to capture knowledge effectively from the domain expert. Therefore, it is important that we improve the versatility of Mecano by featuring a wider choice of available styles.

The use of separate languages to define constraints on the two interaction styles is also troublesome because neither language is general enough to be extended and applied to other interaction styles that may be added in the future. Furthermore, because PCL allows only connectivity constraints, it is useful only for nodes-and-links diagrams. The solution that we are pursuing is to develop a single language with a high degree of generality that can accommodate constraint declaration for multiple interaction styles and diagramming paradigms.

Mecano is highly adaptable to individual tasks and domains. Adaptation to individual users, however, is limited to modifying the layout of forms, or changing the appearance of the palette components. We are exploring the possibility of enforcing constraints in different ways for different users. For example, the suggested and illegal

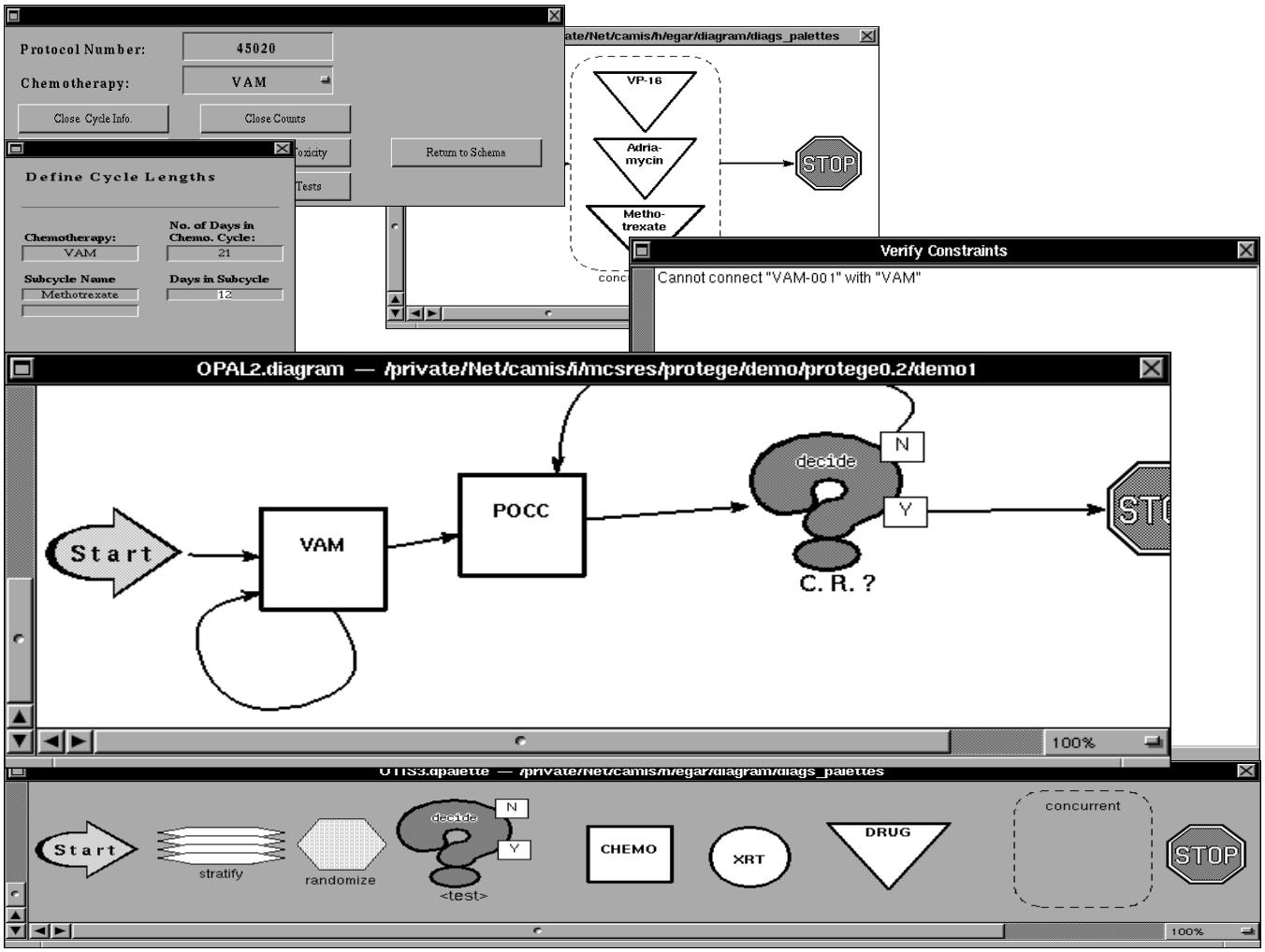

Figure 4: The knowledge-editing phase. Mecano manages the editing sessions by conducting constraint verification and by coordinating the display of linked interface components.

paths established in the knowledge editor by the knowledge engineer could change for users who have different levels of expertise in the use of the knowledge editor. The suggested path would be more flexible (fewer visual constraints) for an experienced user and more rigid for a novice, whereas the illegal path would be extensive (more connectivity constraints) for a novice and more constricted for an experienced user.

CONCLUSIONS

We have presented Mecano, a user-interface management system that generates knowledge editors for the PROTÉGÉ-II knowledge-acquisition shell independently of the underlying problem-solving method selected for task modeling. Mecano not only provides a general framework for the construction of knowledge editors for multiple methods—a previously unavailable feature in knowledge-acquisition tools—but also automates the construction process and reduces the task of creating a new editor to one of specifying the components of the editor and the relationships among those components.

Mecano also overcomes a deficiency of previous method-oriented knowledge-acquisition tools—namely, the dependency of the interface of the knowledge editor on the computational demands of the domain-independent problem-solving method on which the tools are based. Mecano generates knowledge editors that are also based on a selected method, but that are adaptable to the needs and requirements of the task and domain for which knowledge is acquired, and to those of the users of the knowledge editor.

Through the use of constraints and the declaration of display dependencies among interface components, knowledge editors generated by Mecano guide the domain expert during the editing sessions, providing visual hints of what actions are expected of the user, and disallowing

operations on the interface components that are invalid within the context of the current task.

The success of Mecano serves as a clear indication of how certain fields, such as knowledge acquisition, can benefit from human–computer interaction research, and how the solution to some of the field's key challenges can best be obtained by increasing the focus on interface issues that had previously taken a back seat to the more traditional knowledge-representation and knowledge-modeling problems.

ACKNOWLEDGMENTS

This work has been supported in part by grants LM05157 and LM05208 from the National Library of Medicine, by grant HS06330 from the Agency for Health Care Policy and Research, and by a gift from Digital Equipment Corporation.

We thank Andrew Bennett for his work in developing FormIKA, and Lyn Dupré for her extensive editing of a previous draft of this paper.

REFERENCES

1. Bennett, A. 1990. *A form-based user interface management system for knowledge acquisition.* Master's Thesis. Report KSL-90–43, Knowledge Systems Laboratory, Stanford University, Stanford, CA.
2. 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.
3. Clancey, W. J. 1985. Heuristic classification. *Artificial Intelligence* **27**:289–350.
4. 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.
5. Friedland, P. E., and Iwasaki, Y. 1985. The concept and implementation of skeletal plans. *Journal of Automated Reasoning* **1**:161–208.
6. Klinker, G., Bhola, C., Dallemagne, G., Marques, D., and McDermott, J. 1991. Usable and reusable programming constructs. K*nowledge Acquisition* **3**:117–135.
7. Marcus, S. and McDermott, J. 1989. SALT: A knowledge acquisition tool for propose-and-revise systems. *Artificial Intelligence* **39**:1–37.
8. Musen, M.A. 1989. *Automated Generation of Model-Based Knowledge-Acquisition Tools.* London: Pitman.
9. Musen, M. A. 1989. An editor for the conceptual models of interactive knowledge-acquisition tools. *International Journal of Man-Machine Studies* **31**:673–698.
10. Musen, M.A., and Tu, S.W. 1991. *A model of skeletal-plan refinement to generate task-specific knowledge acquisition tools.* Report KSL-91–05, Knowledge Systems Laboratory, Stanford University, Stanford, CA.
11. Puerta, A.R., Egar, J.W., Tu, S.W., and Musen, M.A. In press. A multiple-method knowledge-acquisition Shell for the automatic generation of knowledge-acquisition tools. In *Proceedings of the Sixth Knowledge-Acquisition for Knowledge-Based Systems Workshop*, Boose, J.H., and Gaines, B.R., editors. Alberta, Canada.
12. Steels, L. 1990. Components of expertise. *AI Magazine* **11**(2):30–49.