Generation of Knowledge-Acquisition Tools from Reusable Domain Ontologies

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

Medical Computer Science Group
Knowledge Systems Laboratory
Stanford University
Stanford, California, 94305-5479, USA
Telephone: (415) 723-6979
E-mail: {puerta,eriksson,egar,musen}@camis.stanford.edu

Abstract

We present Mecano, a development environment that automates the design of knowledge-acquisition software tools for knowledge-based systems. Mecano is a component of PROTÉGÉ-II—a development environment for knowledge-based systems. Given an explicit model of the domain—called a *domain ontology*—that is shared by Mecano and the target knowledge-based system, we show how a specification for a knowledge-acquisition tool can be inferred. In particular, the domain ontology is used to determine (1) the knowledge requirements of the target knowledge-based system that must be fulfilled by the knowledge-acquisition tool, (2) the appropriate dialog sequences with users of the tool that ensures the consistency and completeness of the acquired knowledge, and (3) the presentation and layout of the components of the tool's interface. The use of shared domain ontologies to design knowledge-acquisition tools facilitates the integration of tools with target knowledge-based systems, produces domain-oriented tools, and eases the propagation of changes in the domain ontologies from the knowledge-based systems to the knowledge-acquisition tools.

1. Tools as Interfaces

The development of *knowledge-acquisition tools*—systems that allow either knowledge engineers or domain experts to edit knowledge bases—presents a challenge to developers of knowledge-based systems. Some of the requirements that make these tools difficult to implement are:

- High-level integration with a target knowledge-based system
- Complex user interface with sophisticated *dialogs*[1]
- Strong orientation toward a single domain and task
- Limited number of users, sometimes as few as one

1. A dialog is the sequence of actions that an interface allows a user to perform.

These requirements dictate that knowledge-acquisition tools are often useful for only a single knowledge-based system, or, at most, for a small class of systems based on a common problem-solving method. Developers of these tools generally emphasize the automation of the dialog sessions with the users, and the translation of the results of those dialogs into a proper format in the knowledge base. As a result, knowledge-acquisition tools can, by themselves, be intricate knowledge-based systems, thus creating a situation where the development of one knowledge-based system (e.g., an expert system) necessitates the construction of another one (i.e., the knowledge-acquisition tool). On the other hand, little or no emphasis is placed on the automation of the design of the tool. To design a new knowledge-acquisition tool, knowledge engineers painstakingly examine the knowledge requirements and data structures of the target knowledge-based system, devise dialog sequences, decide on presentation modalities, and, finally, build tools that conform to these design specifications.

In this paper, we emphasize automated design of knowledge-acquisition tools. We argue that these tools should not be viewed as independent programs integrated with other knowledge-based systems. Rather, they should be viewed as interfaces that permit access to the data structures of the knowledge-based system and that present those structures to a user in a manner that ensures the consistency and completeness of the acquired knowledge. The problem of designing a tool therefore can be depicted as that of writing an interface specification. We show how, from an explicit structured collection of domain terms and their interrelationships, called a *domain ontology*, programs can infer such interface specifications by applying dialog and layout rules. These domain ontologies are shared by knowledge-acquisition tools and their target knowledge-based systems, thus harmonizing the integration of both types of systems. Domain ontologies are *reusable knowledge components* [Musen, 1992; Neches et al., 1991], and are useful across different applications.

The knowledge contained in a domain ontology is the basis for the generation of interface specifications for knowledge-acquisition tools. In particular, from this knowledge we can infer (1) the knowledge requirements of the expert system that must be fulfilled by users of the knowledge-acquisition tool, (2) the dialog sequences most appropriate for user interaction, (3) the components of the interfaces and their layout, and (4) the constraints that affect the behavior of the interface. The generation of knowledge-acquisition tools in our work takes place within Mecano—a development environment for knowledge engineers that provides a collection of software tools to generate interface specifications, to custom-tailor specifications, to produce and manage knowledge-acquisition tools from interface specifications, and to maintain knowledge bases. The generation of knowledge-acquisition tools in Mecano provides the following benefits:

- Automatic integration of the tool with the knowledge-based system
- Generation of domain-specific tools, and, when user preferences are included, user-specific tools
- Tool interfaces that conform to user-interface guidelines
- Ease of propagation of domain-ontology changes into the knowledge-acquisition tool

2. Related Work

There are many examples of knowledge-acquisition tools developed in tandem with expert systems, but without automation of the design process. These examples range from early tools such as TEIRESIAS [Davis, 1979], to more recent ones such as ROGET [Bennett, 1985] and SALT [Marcus and McDermott, 1989]. The introduction of *metatools*, such as PROTÉGÉ-I [Musen, 1989] and DOTS [Eriksson, 1991], marked a departure from completely manual design. PROTÉGÉ-I generates tools directly from a role-limiting problem-solving method [McDermott, 1988]. It is restricted to a single method, and, because problem-solving methods are domain independent, the interfaces of the resulting tools follow the computational requirements of the method, and ignore the interaction requirements of the domain of interest and of the tool users [Puerta et al., 1992]. DOTS is not limited to any specific problem-solving method; its function is to *assist* knowledge engineers in the design of new tools, rather than to automate the design process.

Researchers in human–computer interaction are studying the generation of user interfaces from data models. Systems such as HUMANOID [Szekely et al., 1992] and UIDE [de Barr et al., 1992] use the data models of an application to produce a user interface for that application. The level of automation, however, is limited by the expressiveness of the data models. A particular shortcoming is the inability of these models to represent relationships among the different data objects in the application. We shall show that these relationships—which are explicit in the domain ontologies used by Mecano—are crucial in the generation of dialog sequences for knowledge acquisition. The remainder of this paper is organized as follows. Section 3 presents an overview of Mecano. Section 4 explains how, from a domain ontology, Mecano generates form-based knowledge-acquisition tools. Section 5 contains conclusions and discusses open research questions.

3. The Mecano Environment

The Mecano development environment is part of the PROTÉGÉ-II architecture [Puerta et al., 1992; Puerta et al., in press], shown in Figure 1, that is under construction at our laboratory. PROTÉGÉ-II allows knowledge engineers to construct expert systems from a library of

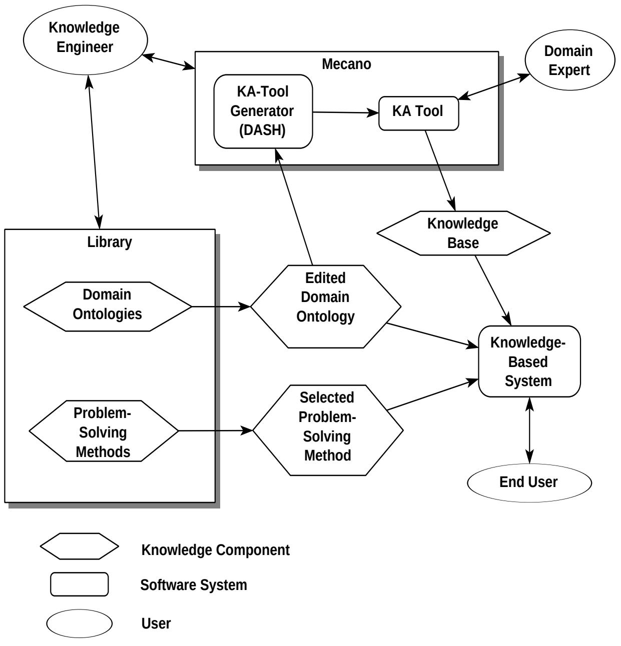

Figure 1. A schematic view of PROTÉGÉ-II. To build a knowledge-based system, the developer selects a domain-independent problem-solving method from a library and, by editing a domain ontology as required by the chosen method, maps this method to a given domain. The developer then uses the edited domain ontology to generate an appropriate knowledge-acquisition tool. A domain expert uses this knowledge-acquisition tool to edit a knowledge base. The knowledge-based system uses the selected problem-solving method to reason about the knowledge base.

problem-solving methods, thus overcoming the single-method limitation of PROTÉGÉ-I. The domain-independent methods in the library are reusable and can be combined to create other methods, thereby supplying the building blocks for a broad range of applications [Eriksson et al., 1992]. If a method is to be applied to a given domain, the method must be *mapped* to that domain through a domain ontology [Neches et al., 1991]. For our purposes, a domain ontology is an explicit structured collection of domain terms (e.g., a class hierarchy) and their interrelationships. Domain ontologies are stored in PROTÉGÉ-II in the same library system that contains the problem-solving methods. The library of PROTÉGÉ-II provides the facilities that allow knowledge engineers to index and search ontologies and methods, to combine methods to create new ones, and to map methods to domains by editing ontologies.

A problem-solving method contains control knowledge that is related (mapped) to a domain through a domain ontology. Such mapped control knowledge guides the reasoning process of knowledge-based systems built with PROTÉGÉ-II. There is also, however, a need to acquire the propositional domain knowledge on which the target knowledge-based system will operate. In PROTÉGÉ-II, this knowledge is acquired directly from domain experts through knowledge-acquisition tools that are generated from the domain ontology. Mecano provides a development environment that allows knowledge engineers to generate such tools with a minimum of custom-tailoring and manual design. The knowledge-acquisition tools are viewed as interfaces that (1) allow users to fulfill the knowledge requirements of the problem-solving method, (2) conduct dialogs with users (i.e., domain experts) in a manner that ensures the completeness and consistency of the knowledge acquired, and (3) present information to users in a style that is natural to the latter. The last two points are particularly important when domain experts are not sophisticated computer users.

Mecano functions as an integrated collection of software tools to generate knowledge-acquisition tools from domain ontologies. Figure 2 depicts the various types of tools and components available in this environment. Eventually, Mecano will support knowledge-acquisition tools with multiple interaction styles, such as combinations of form-based and graph-based editors. Separate sets of software tools are required to implement each interaction style. The components shown in Figure 2 have the following features:

- *Interface-specification generators* input a domain ontology; determine the dialog and layout characteristics of the interface, according to a knowledge base of dialog and layout rules; and output an interface specification.

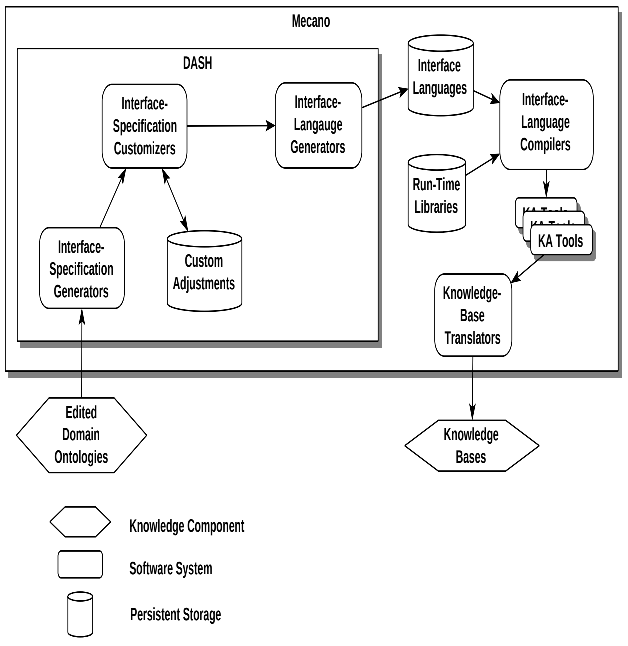

Figure 2. The Mecano development environment. Interface specifications are generated from domain ontologies. The specifications may be custom-tailored by the knowledge engineer according to user preferences. The custom-tailored specifications are converted into textual specifications in one or

- *Interface-specification customizers* allow knowledge engineers to refine the interface specification and correct possible shortcomings of the generated specification.
- *Interface-language generators* translate high-level interface specifications into declarative *interface languages* that can be linked with *run-time libraries,* by *interface-language compilers,* to produce *knowledge-acquisition tools.*
- *Knowledge-base translators* take the knowledge representation acquired in the knowledge-acquisition tools (e.g., a graph, or form data) and translate it into an appropriate format for the knowledge base of the target application of PROTÉGÉ-II.

In this paper, we concentrate on the generation of interfaces from domain ontologies. The conceptualization and use of a declarative interface language for forms has been reported elsewhere [Bennett, 1990], as has the interpretation and translation of graphs into a textual knowledge representation [Egar, Puerta, and Musen, 1992].

4. Generation Of Tools From Ontologies

In this section, we discuss a component of Mecano called DASH (Figure 2). This subsystem implements functionality for interface-specification generation and custom-tailoring, as well as for interface-language generation. The interaction style supported by DASH is form-based knowledge acquisition.

By using domain ontologies as input to the interface specification generators, and by viewing knowledge-acquisition tools as user interfaces, we can approach the problem of designing a new tool systematically. In DASH, the design of a form-based tool is broken down into the two principal design steps of a user interface: dialog design, and layout design. The remainder of this section details how the domain knowledge from the input domain ontologies is applied to complete each of the two design steps.

4.1 Generation of Top-Level Dialog Structures

The first step in generating a knowledge-acquisition tool is to design the top-level *dialog structure* of the tool. In DASH, the dialog structure is a nodes-and-links graph that describes the nature of the knowledge-acquisition tool from the user's perspective. The dialog structure defines what is the overall organization of a graphical user interface, and how users can move among different components of the user interface. The nodes in the graph represent interface components such as menus, browsers, or knowledge-editing windows (referred to as *knowledge editors)*. The links define the ways in which the user can access these components. For instance, a knowledge-acquisition tool may have a main menu that provides access to several browsers, which allow the user to edit sets of objects using knowledge editors.

DASH uses the relationships among the classes defined in the ontology to design a dialog structure for the target knowledge-acquisition tool. Thus, the definitions that DASH uses as a basis for tool design are the same as those that the problem solver—the problem-solving method of the target knowledge-based system—uses for reasoning. DASH analyzes the network that these relationships form, and creates a dialog structure where knowledge editors operate on objects of the ontology, and where users can create and browse objects according to their ontology definitions.

A subsystem of DASH—the *dialog designer*—generates a dialog structure, which provides the basis for subsequent generation of a complete knowledge-acquisition tool, from the input ontology. The algorithm that DASH uses for transforming the ontology to a dialog structure consists of the following steps:

1. Index the class definitions and the relationships among them.
2. For all classes defined, create a user-interface prototype component.

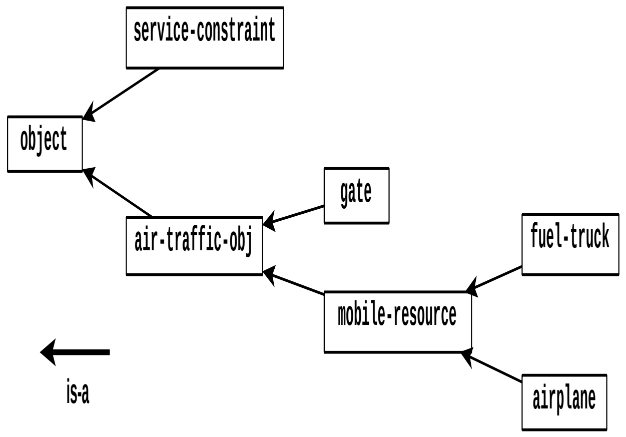

Figure 3. A sample ontology for the airport domain. Given this input ontology, the dialog designer produces a dialog structure for the knowledge-acquisition tool.

3. For all component prototypes, link the prototypes according to the relationships among the classes of the ontology (i.e., according to the slot types).
4. Add the user-interface components required to access all components of the emerging graph (e.g., main menu and browsers) according to a set of *dialog rules* based on guidelines for user-interface design.
5. Optimize the graph representing the dialog structure by applying refinement rules (e.g., combining two small windows to one larger window by merging two nodes representing the initial windows), and instantiate the prototypes to operational windows, menus, browsers, and so on.

We shall illustrate how the dialog designer operates by providing an example of the generation of dialog structures in DASH. Figure 3 shows a sample ontology from a scheduling problem in an airport domain; the task is to coordinate the scheduling of airplanes, gates, and fuel trucks. The most general class, `object`, has the subclasses `service-constraint` and `air-traffic-obj`. The class `air-traffic-obj` has the subclasses `gate` and `mobile-resource`. The latter subclass represents vehicles at the airport—such as airplanes and fuel trucks. Figure 4 shows the dialog structure generated by the dialog designer. In this dialog structure, the main menu provides access to three list browsers that manage sets of `service-constraints`, `fuel-trucks`, and `gates`. The arrows in the dialog structure represent the accessibility relationships among the user-interface components. The abstract classes `object`, `air-traffic-obj`, and `mobile-resource` are not reflected in the dialog structure because they cannot be instantiated; it is pointless to provide editors for these classes. In

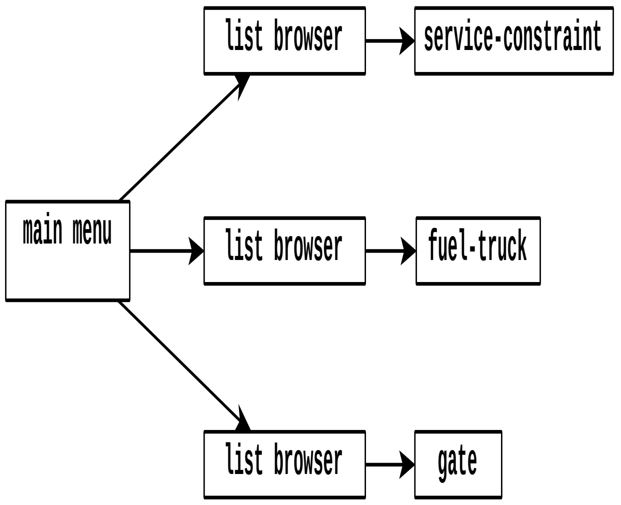

Figure 4. A top-level dialog structure generated for the airport ontology by the dialog designer.

addition to the user-interface components originating from classes (i.e., `service-constraint`, `fuel-truck`, and `gate`), step 4 of the generation algorithm ensures that the dialog designer adds browsers for accessing these components and a main menu that manages the target knowledge-acquisition tool and that provides access to the browsers.

4.2 Generation of Layouts for Form-Based Tools

Forms are an important type of knowledge editor. Forms can be used by domain experts to enter and edit relatively structured information. The *layout designer* of DASH produces window and form layouts based on the dialog structure and on additional information in the class definitions (e.g., slot types). Normally, DASH produces a form for each form-based knowledge-editor node in the dialog structure—that is, DASH creates a form for each class definition. The layout designer uses a layout algorithm that derives from the domain ontology the interface components (e.g., text fields and check boxes) that make up each form. Then, it lines up the user-interface components according to the slot definitions in the corresponding class. Unfortunately, automated layout algorithms have one central common problem: the design space is exceedingly large and is influenced by many factors, such as user preferences, that are difficult to anticipate [Szekely et al., 1992]. Therefore, the goal of the layout algorithm is not to generate a perfect layout for the end user, but rather to generate a layout that provides a good starting point for custom adjustments by the developer. In this approach, the developer uses the prototype layout that DASH generates as a canvas. Custom-tailoring of the window layout is achieved by direct manipulation in a graphical tool. For instance, the developer can modify default labels generated from slot names, and can reposition the fields on a form.

User-interface widgets provided by commercially available window systems, such as X Window and NeXTStep, can be abstracted to *selectors* [Johnson, 1992]. Selectors model user tasks in the user interface (e.g., selecting a particular item from a list of items). Selectors can be instantiated to widgets at design time, or at run time. DASH uses selectors as an intermediate design step in the generation of forms. Table 1 shows the mapping from slot types to selectors and widgets that DASH uses to build form layouts. A slot of type integer, for example, maps into a numeric selector, which can be instantiated to a numeric field widget. DASH uses the preconditions on these mapping rules to select among several widget instances. For example, the enumeration type can be instantiated as widgets for radio buttons, or as a single pop-up menu widget.

Table 1. The mapping from data types to selectors and potential widget instances.

slot data type	selector	widget instance
integer	numeric	numeric field
real	numeric	numeric field
string	string	text field
boolean	settings	check box
		toggle button
enumeration	settings	radio buttons
		pop-up menu

We shall illustrate the generation of forms from class definitions by providing an example of a form for airport gates. In this example, the class `gate` contains the slots `gate-number`, `passenger-limit`, and `international`. The slots `gate-number` and `passenger-limit` are of type `integer`; the slot `international` is of type `boolean`. Figure 5 shows the resulting form. Numeric fields have been generated for the number and the passenger limit of the gate, and a check box has been generated for the international flag. The developer can custom-tailor the form using direct manipulation. DASH produces as output a description of the target knowledge-acquisition tool in the form-definition language FormIKA [Bennett, 1990], which can be compiled into C code for efficient execution (see Figure 2). Domain experts can then use the forms to enter and edit domain knowledge.

The PROTÉGÉ-II environment encourages developers to build knowledge-based systems incrementally. Often, the

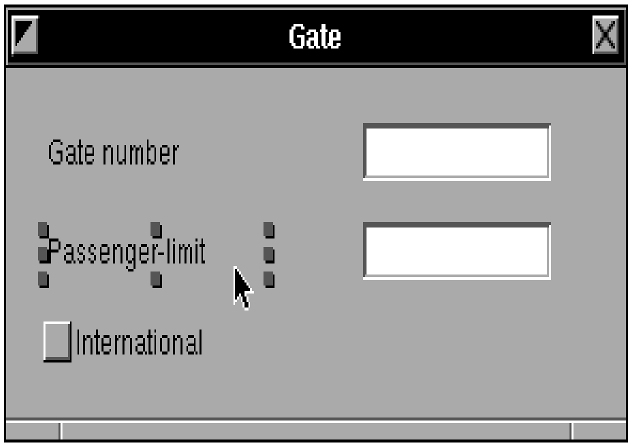

Figure 5. The generated form for gates. The developer can adjust the form layout through direct manipulation.

domain ontology is modified and extended several times during the development process. Such modifications and extensions create a maintenance problem in that the knowledge-acquisition tool generated must be updated to support the new class definitions. Preserving manual adjustments to the knowledge-acquisition tool over regenerations of the tool presents additional challenges. DASH supports *persistent custom adjustments*—that is, DASH preserves the custom adjustments made by the developer in a custom-tailoring database. For instance, the developer can add slots to a class definition, and can use DASH to regenerate the knowledge-acquisition tool. DASH preserves the original custom layout of the form generated for this class, and adds fields for the new slots to the form. Thus, the developer can use DASH to generate an initial knowledge-acquisition tool from an early version of the domain ontology, and can continue refining the knowledge-acquisition tool as the ontology evolves. DASH implements persistent custom adjustments by managing the custom-tailoring database. When custom adjustments from previous sessions exist in the database, DASH reapplies these adjustments before presenting them to the developer and allowing additional changes.

5. Discussion

Traditionally, developers of knowledge-based systems have had to design and build new knowledge-acquisition tools each time their target knowledge-based system employed a new problem-solving method [Bennett, 1985; McDermott, 1988; Marcus and McDermott, 1989]. The automation of the design of knowledge-acquisition tools that is offered by Mecano offers important benefits for the developers of knowledge-based systems. Two clear advantages are (1) the direct integration of knowledge-acquisition tools with the target knowledge-based systems; and (2) the ease of propagation of changes to the domain model employed by the knowledge-based system, into the knowledge-acquisition tool. In particular, the consistency of changes to the domain models is ensured by the use of domain ontologies that are shared by knowledge-acquisition tools and their target knowledge-based systems. Mecano combines commitments of the domain ontologies and of the problem-solving methods to create the knowledge-acquisition tool and its user interface. The obvious reduction in the developer's programming burden affords Mecano developers the freedom to produce a separate knowledge-acquisition tool for each task and domain—and potentially for each individual user—regardless of whether the underlying problem-solving method changes.

The use of domain ontologies in Mecano is consistent with the framework of *reusable knowledge components* [Musen, 1992; Neches et al., 1991]. In this approach, the developer builds knowledge-based systems by assembling them from several reusable problem-solving constituents that operate on a common domain ontology. Within this framework, the knowledge-acquisition tools that fulfill the knowledge requirements of the problem-solving components may be developed by either (1) associating a generic knowledge-acquisition tool with each problem-solving component, or (2) associating a domain-specific tool with each task to be performed by the problem-solving component. The first option, which is advocated in systems such as Spark [Marques et al., 1992], has the disadvantage that it does not produce domain-specific knowledge-acquisition tools; such specificity is an important requirement for many tools, especially with regard to the definition of domain-specific dialogs with users of the tools. Furthermore, the tight coupling between a single component and a single tool, further complicates the construction of systems that encompass multiple problem-solving methods. In such systems, developers must associate a knowledge-acquisition tool that combines the features of the tools associated with each of the individual problem-solving methods.

In PROTÉGÉ-II, we follow the second option. Mecano generates knowledge-acquisition tools for systems that are built from a library of problem-solving methods. PROTÉGÉ-II relies on Mecano to generate domain-specific tools. This approach allows knowledge engineers to produce a knowledge-acquisition tool that insulates the knowledge engineer from the structure of the problem solver, and that provides a coherent conceptual model to the expert, even if the problem solver is composed from multiple problem-solving methods. The usefulness of Mecano is not necessarily limited to knowledge-based systems built with PROTÉGÉ-II. Given ontologies from sources other than PROTÉGÉ-II, we can potentially translate those ontologies from a standard interchange format [Neches et al., 1991], and can incorporate them into Mecano.

The role of environments such as Mecano is to assist the developer in the design of knowledge-acquisition tools from ontologies. This task can be seen as a mapping from domain ontologies to user-interface ontologies of the target tools. Because this mapping can be custom-tailored, the developer can define several alternative mappings—that is, alternative target knowledge-acquisition tools for the same domain ontology. In this scenario, it is important that changes to the common domain ontology do not result in a need to *remap* all the alternative user-interface ontologies that may have been produced before the changes. Therefore, Mecano treats custom adjustments as persistent mappings that can be reapplied automatically with each new version of the domain ontology. We estimate that it is possible to implement similar mappings from ontologies to other target software. For instance, it might be possible to use a tool similar to Mecano for the automatic generation of database interfaces.

Through Mecano, we have found that domain ontologies can direct the generation of knowledge-acquisition tools. We believe that Mecano presents a framework where knowledge-acquisition tool design follows directly and automatically from the design of the corresponding knowledge-based system.

Acknowledgments

This work has been supported in part by grants LM05157 and LM05305 from the National Library of Medicine, and by a gift from Digital Equipment Corporation. Dr. Musen is recipient of NSF Young Investigator Award IRI–9257578.

We thank John Gennari, Yuval Shahar, Samson Tu, and Eckart Walther for their insightful comments and discussions, which have helped us to develop the concept and design of Mecano. We are also grateful to Lyn Dupré for editing a previous version of this paper.

References

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.

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

Davis, R. 1979. Interactive transfer of expertise: Acquisition of new inference rules. *Artificial Intelligence*, **12**(2):121–157.

de Baar, D., Foley, J.D., and Mullet, K.E. 1992. Coupling application design and user interface design. In *Proceedings of CHI '92.* Bauersfeld, P., Bennett, J., and Lynch, G., editors, pp. 259–266. Monterey, California, May 1992.

Egar, J.W., Puerta, A.R., and Musen, M.A. 1992. Automated interpretation of diagrams for specification of medical protocols. In *Proceedings of AAAI Spring Symposium on Reasoning with Diagrammatic Representations*. Narayanan, N.H., editor, pp. 189–192. Stanford, California, March, 1992.

Eriksson, H. 1991. *Meta-Tool Support for Knowledge Acquisition.* Ph.D. Dissertation No. 244. Department of Computer and Information Science. Linköping University, Linköping, Sweden.

Eriksson, H., Shahar, Y., Tu, S.W., Puerta, A.R., and Musen, M.A. 1992. Task modeling with reusable problem-solving methods. In *Proceedings of the Seventh Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Boose, J. H. and Gaines, B. R., editors, pp. 8.1–8.24. Banff, Alberta, Canada, October 1992.

Johnson, J. 1992. Selectors: Going beyond user-interface widgets. In *Proceedings of CHI '92.* Bauersfeld, P., Bennett, J., and Lynch, G., editors, pp. 273–279. Monterey, California, May 1992.

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

Marques, D., Dallemagne, G., Klinker, G., McDermott, J., and Tung, D. (1992). Easy programming: Empowering people to build their own applications. *IEEE Expert,* **7**(3), 16–29.

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

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

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

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

Puerta, A.R., Egar, J.W., Tu, S.W., and Musen, M.A. 1992. A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. *Knowledge Acquisition,* **4**(2):171–196.

Puerta, A.R., Tu, S.W., and Musen, M.A. (in press). Modeling tasks with mechanisms. *International Journal of Intelligent Systems*.

Szekely, P., Luo, P., and Neches, R. 1992. Facilitating the exploration of interface design alternatives: The HUMANOID model of interface design. In *Proceedings of CHI ‘92*. Bauersfeld, P., Bennett, J., and Lynch, G., editors, pp. 507–515. Monterey, California, May 1992.