Custom-Tailored Development Tools for Knowledge-Based Systems

Henrik Eriksson* Angel R. Puerta John H. Gennari
Thomas E. Rothenfluh† Samson W. Tu Mark A. Musen

Section on Medical Informatics
Knowledge Systems Laboratory
Stanford University School of Medicine
Stanford, California 94305-5479, U.S.A.

October 2, 1994

Abstract

PROTÉGÉ-II is a development environment for knowledge-based systems. PROTÉGÉ-II supports developers by providing a series of development tools. DASH, which is part of the PROTÉGÉ-II tool set, is a metalevel tool that uses domain ontologies (which are models of domain concepts and relationships among concepts) as the basis for generating domain-specific knowledge-acquisition tools. Domain experts use the tools that DASH generates to enter the knowledge required for problem solving. DASH generates target tools in a series of design steps, and it uses sets of design rules as the basis for the transitions among the design stages.

1 Introduction

During the 1980's, knowledge-based systems were heralded as a new type of software designed to provide expert advice to end users. However, the development of knowledge-based systems that are adequate and usable for the end users has proven more difficult than expected initially. Many researchers and practitioners have highlighted the problem of insufficient integration with other, preexisting software components. Although developers can solve many of these problems by careful system design, the construction of knowledge-based systems is often costly, and the problem-solving performance of the systems may be insufficient for their tasks. Even problems that are relatively simple for human experts may require complex software

*Present address: Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden
†Present address: Psychologisches Institut der Universität Zürich, FG Psychologische Methodenlehre, Zürichbergstr. 43, CH-8044 Zürich, Switzerland

solutions. Developers require adequate tools to build such complex knowledge-based systems and to maintain these systems.

Knowledge-acquisition tools are programs that elicit knowledge directly from domain experts, and produce knowledge bases, which are operationalized by a *problem-solving method* [13]. Ideally, knowledge-acquisition tools should be general to provide support for knowledge acquisition in many application domains. In many cases, however, general knowledge-acquisition tools provide insufficient support, because these tools are too general with respect to the application domain. Domain-specific knowledge-acquisition tools can provide adequate support, but are often too costly to implement for a single development project. Metatools can make domain-specific tools feasible economically by reducing significantly the cost of developing new domain-specific tools [2].

PROTÉGÉ-II is a development environment and a suite of tools that supports the development of knowledge-based systems and software systems with knowledge-based components [18]. The assistance provided by PROTÉGÉ-II is twofold: PROTÉGÉ-II supports the development of problem solvers from reusable components [4], and generates automatically custom-tailored knowledge-acquisition tools for the development of knowledge bases [3]. The development of problem solvers from reusable components is still the topic of ongoing research, but the generation of knowledge-acquisition tools is emerging currently as a practical technology.

The first generation of experimental metalevel tools [14, 1] for generation of knowledge-acquisition tools was difficult to use in practical development projects, because of the detailed tool specification required, and because of the limited generality of the metatool. A new generation of metatools, however, provide viable solutions. The PROTÉGÉ-II project shows that the generation of specialized knowledge-acquisition tools is now maturing as a technique for developing knowledge-based systems. We have used PROTÉGÉ-II's tool generator, DASH, to produce knowledge-acquisition tools for several projects in our laboratory. In addition, other sites are using DASH for tool generation in their development work. We shall discuss the principles for automated tool generation, and shall examine the techniques that DASH uses to generate knowledge-acquisition tools.

2 Approaches to Tool Generation

Automatic tool generation requires a specification for the target tool. The structure of this specification can vary [2]. We shall discuss briefly three approaches to tool specification and generation.

In the *method-oriented* approach, the metatool uses an instantiation of a problem-solving method for a domain as the basis for the tool generation. PROTÉGÉ-I [14] and Spark [12] are examples of method-oriented metatools for generation of knowledge-acquisition tools. A disadvantage of this approach is that the metatool cannot generate knowledge-acquisition tools for domains that require other problem-solving methods.

The *architectural* approach uses an architectural model of the target tool as the basis for the tool specification. The developers provide detailed specifications for the components of the target tool, such as the user interface, internal knowledge representation, and knowledge-base generator. DOTS [1] and SIS [8] are examples of metatools that are based on the architectural approach. The architectural approach is more general than the method-specific approach, but it requires the developer to design the tool in detail and to define the tool components (a

task that can be time consuming).

In the *ontology-based* approach, the metatool generates a knowledge-acquisition tool from an ontology by mapping ontology definitions to user-interface elements in the tool design. PROTÉGÉ-II and DASH use the ontology-based approach. In the PROTÉGÉ-II framework, the input to the metatool is typically an application ontology, but the developer can use other types of ontologies as input to generate other types of knowledge-acquisition tools (such as method ontologies to generate method-specific tools). The ontology-based approach is more general than the method-specific one (in the sense that the space of possible target tools is much larger), but it is less general than the architectural approach. The principal advantage of the ontology-based approach is that developers can specify knowledge-acquisition tools readily, and that a preexisting ontology can be used as the basis for the specification.

3 The PROTÉGÉ-II Suite of Tools

PROTÉGÉ-II uses the notion of *tasks,* which are models of functionality required, and *problem-solving methods,* which perform tasks. The developer can *decompose* problem-solving methods into *subtasks,* which can be accomplished by other methods or by *mechanisms* (which are methods that the developer cannot decompose further) [18]. In the PROTÉGÉ-II approach, the developer selects a problem-solving method from a library of reusable methods, and configures the method by providing appropriate submethods for the selected method's subtasks. Moreover, the developer defines the mappings required to translate the input and output of the method to input and output of other methods in the application system. The retrieval of problem-solving methods from the library, the organization of the library, and the configuration of reusable problem-solving methods for new tasks are difficult principal obstacles to method reuse, and are subject to ongoing research [7].

In addition to the problem-solving methods, the developer must define the knowledge representation that the problem solvers should operate on. Workers in Artificial Intelligence has adopted the term *ontology* to describe a model of concepts and relationships [17]. Developers can organize an ontology according to classes, slots, and slot properties (slot facets) similar to the approach used in object-oriented modeling. In PROTÉGÉ-II, we use an extension of CLIPS [16] as the basic ontology-definition language.

In the PROTÉGÉ-II approach, we distinguish among *domain, method,* and *application* ontologies. Domain ontologies are models of domain concepts and relationships. These ontologies are reusable potentially across several application programs for the domain. For instance, an inventory program and a design-configuration program may share the same component concepts and relationships. Method ontologies are ontologies that model concepts relevant for a problem-solving method. For instance, the *propose-and-revise* method for configuration problems incorporates the concepts of design constraint, constraint violation, and constraint-violation fix [11]. (The propose-and-revise method proposes a tentative solution and improves it by applying fixes that remove design constraints.) To reuse a method for a domain, such as elevator configuration, the developer must relate concepts in the method ontology to corresponding concepts in the domain ontology. The method can then operate on instances of the domain ontology through a mapping to method-specific instances, which the method is designed for. For example, problem-solving concepts of the propose-and-revise method can be related to concepts relevant for elevator configuration. Typically, the method ontology is domain independent, but method specific.

The PROTÉGÉ-II approach uses the notion of application ontology to describe the specialization of a domain ontology to include method-specific concerns. The application ontology consists of domain-ontology concepts relevant for the application system (e.g., concepts relevant for elevator configuration) and concepts relevant for problem solving (e.g., upgrade fixes for elevator cables). The application ontology is not intended to be reusable across several systems. In the PROTÉGÉ-II approach, the application ontology is the starting point for the generation of knowledge-acquisition tools rather than the domain ontology, because it incorporates the concepts required by the target system.

The PROTÉGÉ-II tool set includes MAÎTRE, which is an interactive editor for ontologies. MAÎTRE allows the developer to define new classes. The developer can define slots of classes, and can edit class and slot facets (i.e., properties such as type and default value). MAÎTRE provides a graphical user interface for the ontology editing, but stores ontologies in a textual format, which is an extension of CLIPS that incorporates additional class and slot facets used in the generation of knowledge-acquisition tools. Developers use MAÎTRE to develop domain, method, and application ontologies.

DASH is a metalevel tool that generates automatically domain-specific knowledge-acquisition tools [3]. DASH takes as input an application ontology developed in MAÎTRE, and produces as output a specification of a knowledge-acquisition tool. DASH allows the developer to custom tailor the interface of the knowledge-acquisition tool, and to store these custom adjustments persistently.

The tool MEDITOR is a run-time system for knowledge-acquisition tools that can read the tool specifications output by DASH, and can provide immediately an executable tool by instantiating the specifications to appropriate user-interface windows and widgets. Figure 1 shows a sample form from a knowledge-acquisition tool generated by DASH and run by MEDITOR.

The output of the knowledge-acquisition tool is a set of instances of the application ontology. For example, the output from a knowledge-acquisition tool for elevator configuration can be instances of concepts, such as *cable, motor, counter weight,* and *design constraint.* To be usable by a problem solver, these instances must be mapped to the method ontology, which the problem-solver understands (e.g., instances of concepts such as *variable, constraint,* and *fix*). MARBLE is a program that translates the instances produced by the knowledge-acquisition tool to appropriate instances for the problem solver [5]. The translation process is guided by a set of mapping-rule definitions provided by the developer. The current implementation of MARBLE is a prototype system, but we are working currently on an extended version of MARBLE that will handle generalized mappings in a more principled way than the current version. Because it is possible to define an ontology of mapping rules, we can generate readily a knowledge-acquisition tool for such mappings by applying DASH to this ontology.

Figure 2 shows the control panel for PROTÉGÉ-II. This panel is designed to provide an overview of the PROTÉGÉ-II architecture, and to allow the developer to access the intermediate data files produced by these tools. In this panel, the tools are represented by icons organized in a circle (to illustrate how design iterations can be performed). Icons for documents and data files are located in the outer circle. The normal work flow is that the developer creates an application ontology in MAÎTRE and generates a knowledge-acquisition tool in DASH, which the domain expert uses to create a knowledge base. The developer then defines the mapping from the application knowledge base to the format required by the problem solver.

Clinical-trial-protocol

Protocol Number: CCTG552

Protocol Title: Pyrimethamine Therapy for Prevention of Toxoplasma

Purpose: Evaluate 150 mg q wk vs 50 mg q wk of pyrimethamine

Investigators (Add / Delete / Edit)
Deresinski
Kemper
Sison

Eligibility criteria (Add / Delete / Edit)
HIV status
age limitation
SGOT
SGPT
CD4 count

Algorithm | Duration | Follow Up

Regimens (Add / Delete / Edit)
group1
group2

Evaluation schedule (Add / Delete / Edit)

Toxicities (Add / Delete / Edit)
grade 3/4 toxicity
anemia

Toxicity management drugs (Add / Delete / Edit)
leukovorin
iron Supplement

Definitions (Add / Delete / Edit)

Figure 1: Sample form from a knowledge-acquisition tool. Physicians use this form to specify clinical-trial protocols for AIDS treatment. MEDITOR manages the data entered in the tool, and allows the tool user to save the data on file, and to generate knowledge bases consisting of instances of the application ontology.

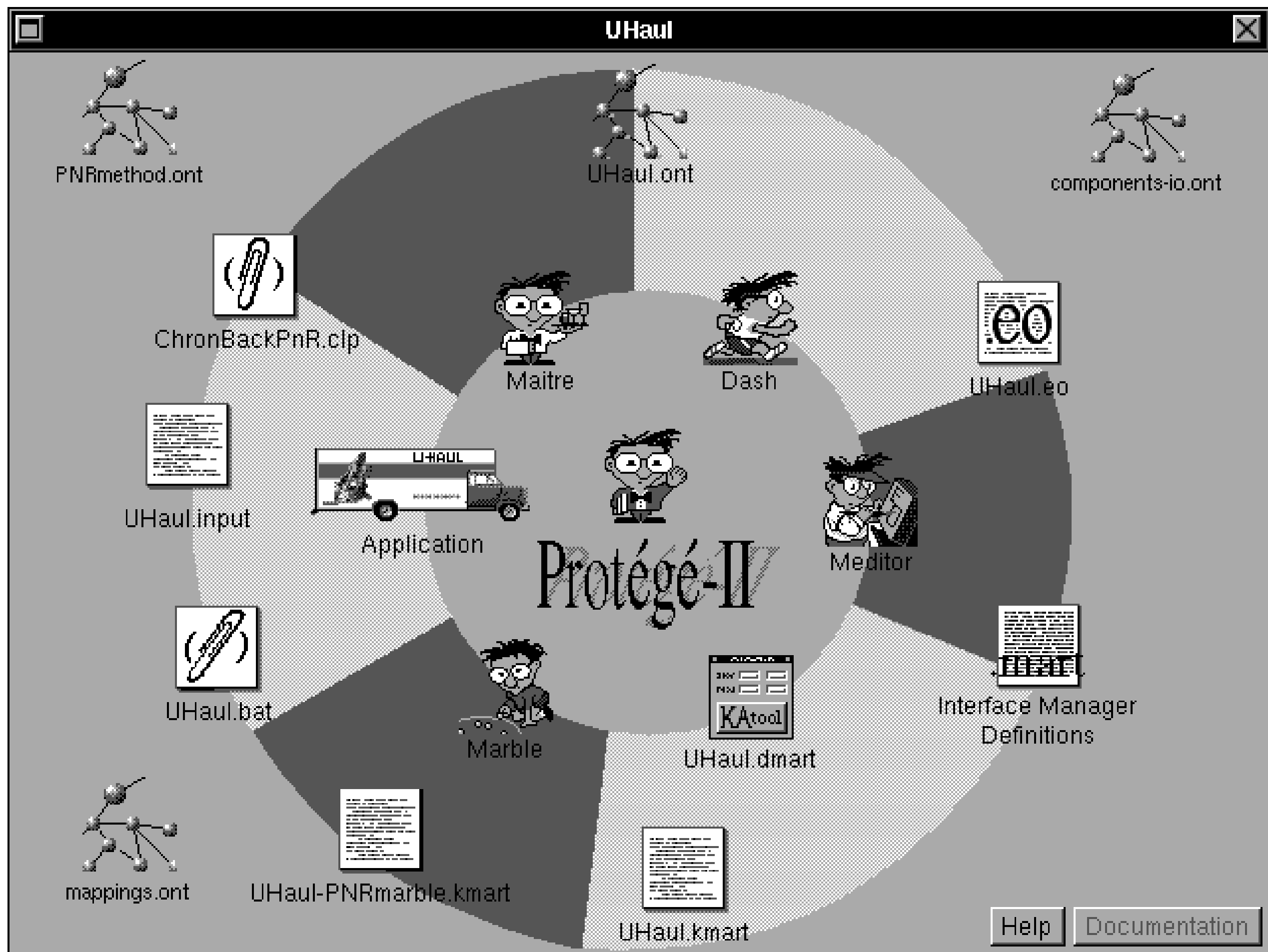

Figure 2: The control panel for PROTÉGÉ-II. The PROTÉGÉ-II user use this panel to invoke tools and to access files produced by the tools.

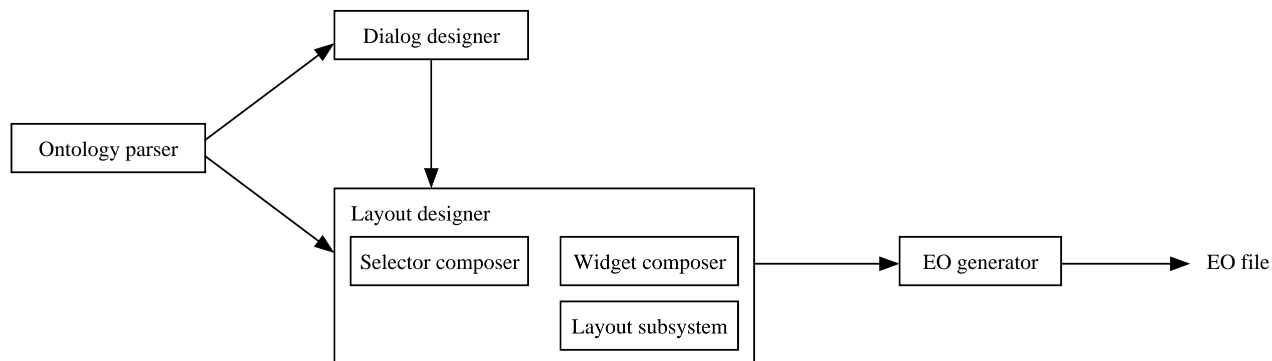

Figure 3: Overview of the DASH architecture. The ontology parser reads the input ontology. The dialog designer analyzes the ontology definitions, and produces the dialog structure. The ontology definitions and the dialog structure are the input to the layout designer. The layout designer consists of the selector and widget composers (which are responsible for the selector and widget designs, respectively) and the layout subsystem for the initial and custom-tailored window layout. The output of DASH is a file of editor-ontology (EO) instances.

4 The DASH Architecture

The purpose of DASH is to automate much of the design task for knowledge-acquisition tools, and to relieve the developer from the burden of dealing with the often intricate details of user-interface implementation for graphical user interfaces. Automated design allows developers new to this type of tool design to build knowledge-acquisition tools with minimal training, and allows developers with extensive experience at tool design to produce tools precisely and rapidly. Automated detailed implementation reduces the number of bugs, and, therefore, the testing, debugging, and maintenance costs for the tool.

We shall describe briefly the major components of the DASH architecture and the design steps involved in tool generation in the DASH approach. After analyzing the input ontology, DASH designs knowledge-acquisition tools by first generating an initial design automatically, and then allowing the developer to custom tailor the design manually. DASH uses a hierarchical design approach to generate knowledge-acquisition tools. In DASH, the *dialog-designer* module first creates a high-level description of the editors and forms in the tool, and the *layout-designer* module then instantiates these descriptions (Figure 3).

The dialog designer performs an important early design task, because, before DASH can generate forms by mapping ontology definitions to user-interface components, DASH must establish the forms to generate and the relationships among these forms. Moreover, the layout designer must know the access paths from a form to other forms before it can render the controls that will allow the user to access the subforms. Examples of such controls are buttons, browsers, graphs, and pop-up lists of instances. Figure 4 shows a sample *dialog structure* produced by the dialog designer.

The task of the layout designer is to instantiate the dialog structure produced by the dialog designer. The layout designer traverses the dialog-structure graph and produces window definitions for every node in the dialog structure. Furthermore, the layout designer maps ontology definitions, such as slot data types, to appropriate user-interface components using

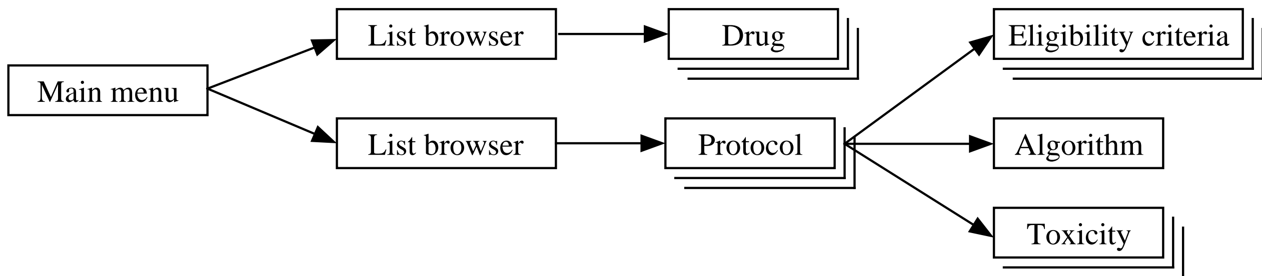

Figure 4: A sample dialog structure for a knowledge-acquisition tool. The main menu provides access to two list-browser windows. These list browsers allow the tool user to create *drug* and *protocol* definitions, respectively. A protocol-definition window contains browsers for *eligibility criteria* and *toxicity,* and provides access to an *algorithm-definition* window.

a set of tool-design rules. As part of the design process, the layout designer maps the slot definitions to *selectors* [6], which are intermediate high-level representations of user-interface widgets. Next, the layout designer instantiates the selectors to widgets, and lays out the widgets on windows. The developer can then custom tailor the layout of the windows.

Custom adjustment is a powerful technique for adapting the user interface of target tools to their users. However, it is often necessary to make changes to the input ontologies after the developers have custom tailored the knowledge-acquisition tools. For instance, sometimes developers discover flaws in ontologies by examining the knowledge-acquisition tools generated from them. If the developers design the target knowledge-based systems and knowledge-acquisition tools incrementally, a series of extensions and modifications to the ontologies will be required. Examples of such ontology changes are additions and deletions of classes and slots in class definitions.

DASH relieves the developer from the burden of readjusting the target tools after changes to the ontologies by supporting *persistent custom adjustments.* DASH stores the changes in a database, and reapplies these adjustments when it regenerates the knowledge-acquisition tools. DASH then allows the developer to make additional manual adjustments to the new version of the target tool. Figure 5 illustrates the use of the custom-tailoring database. DASH installs the adjustments stored in version $n-1$ of the database, and produces, in addition to the knowledge-acquisition tool, a new version n of the database.

When developers change input ontologies, DASH must reapply the custom adjustments that are relevant for the new ontology versions. Examples of such changes are slot additions and deletions, and class additions and deletions. DASH uses class and slot names to match the new ontology version and the records in the custom-tailoring database. As part of this proccess, DASH identifies changes to the target tool caused by added and deleted classes and slots, and takes appropriate actions, such as ignoring custom-tailoring information that correspond to deleted classes and slots.

5 Design Rules

We can view the DASH approach to tool generation as a series of mappings guided by sets of design rules. Although we can sometimes treat the creation of the mappings as a one-step

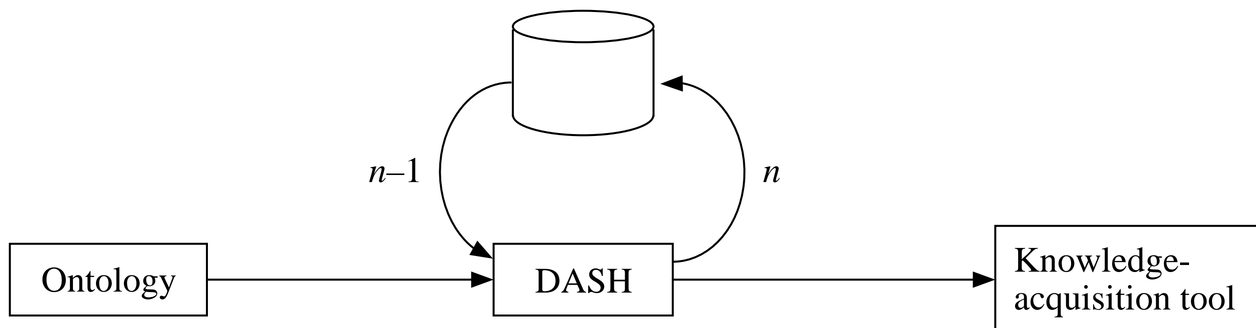

Figure 5: The custom-tailoring database. DASH uses the previous custom adjustments (if any) as the basis for window layout, and stores new custom adjustments for later use (in subsequent regenerations of the target tool).

process, DASH actually first maps the data types to selectors and then maps the selectors to widgets. The selectors are an important intermediate representation in DASH's design process, and they simplify the design rules for target tools.

5.1 Mapping Rules

DASH uses the data types of slot definitions in the input ontology as the basis for the design of appropriate selectors and widgets. The relevant slot data types are *integer, float, string, symbol,* and *instance.* For example, a slot of type integer can be defined as `(slot no-of-employees (type integer))`. Furthermore, the developer can slots of multiple cardinality; that is, lists of data elements. For example, the slot definition `(slot subdivisions (type instance) (cardinality multiple) (allowed-classes division))` specifies a slot that holds a list of instances of the class `division`. In addition to the slot data types, the developer can annotate the ontology definitions with information relevant for tool generation, (e.g., requests for graph editors rather than list browsers).

DASH transforms that have *numeric* data types (integers and floating-point numbers) to entry fields for numbers and appropriate labels based on the slot names. DASH maps numeric slots to numeric-field selections, and then expands the selectors to label and entry-field widgets. DASH uses a similar approach to map slots of type *string* to text-entry selectors, and to label and entry-field widgets.

Slots of type *symbol* are similar to slots of type string, except that certain characters (e.g., space) are illegal in symbol names. Also, slots of type symbol definitions can contain a list of allowed symbols for the slot. If the developer provides a list of allowed symbols, DASH maps the slot to a selection selector (which defines a selection of one item from a list of items), and later to a radio-button widget group *or* to a pop-up widget (which allows the target-tool user to select one of the allowed symbols from the pop-up menu). The number of items to select from determines the choice between radio-button and pop-up widgets. DASH maps slots of type *boolean* to selection selectors that use the items *true* and *false,* and then to check-box widgets.

Let us consider an example of these mappings. Figures 6 and 7 show a sample class definition and the resulting form window produced by DASH, respectively. In the class

```
(defclass corporation (is-a USER)
     (slot company-name (type string))
     (slot no-of-employees (type integer))
     (slot blue-chip (type boolean))
     (slot chapter-11 (type boolean))
     (slot manufacturing-type (type symbol)
          (allowed-symbols discrete process))
     (slot subdivisions (type instance)
          (cardinality multiple)
          (allowed-classes division))
 )
```

Figure 6: The sample definition of the `corporation` class.

`corporation,` the type of the slot `company-name` is *string.* DASH maps this slot to a text-entry selector, and to label and entry-field widgets (as shown in Figure 7). Similarly, DASH maps the slots `no-of-employees, blue-chip, chapter-11,` and `manufacturing-type` to their corresponding selectors and widgets. Note that the slot `manufacturing-type` results in a radio-button group that allows the user to select between *discrete* and *process,* rather than a pop-up menu, because there are only two items to select between.

DASH maps slots of type *instance* to different selectors and widgets depending on the value of other slot facets. If the developer specifies a slot of type instance, cardinality single (i.e., a single instance, not a list of instances), and provides an allowed class for the instance pointed to, DASH maps the slot to a button selector, and to a push-button widget, which provides access to an editor for an instance of the class specified (according to the dialog structure). If the developer specifies a slot of type instance, cardinality multiple, and provides a list of one or more allowed classes for the instance pointed to, DASH maps the slot to a browser selector, and to a list-browser widget with control-button widgets (see Figures 6 and 7). The target-tool user uses this browser to add new instances to the list, and to edit instances in the list (through editors according to the dialog structure). If the developer specifies a slot as type instance, cardinality single, and marks the slot as an *instance pointer,* DASH maps the slot to an instance-pointer selector and to a pop-up widget, which allows the target-tool user to select an instance (of the allowed classes) from a list of instances entered in the target tool.

In the DASH approach, *graph editors* are similar to list browsers in the sense that they provide access to a collection of instances. Graph editors are a sophisticated type of browsers where the items are represented by nodes that can be positioned on the graph canvas. Moreover, the target-tool users can relate the items to each other by creating links among them. If the developer specifies a slot of type instance, cardinality multiple, and requests a graph editor (through a slot facet), DASH maps the slot to a grapher selector, and to a set of widgets that constitutes the grapher. For example, Figure 8 shows the definition of the `workflow` class, which includes the slot `flow-graph`. Note the slot facet `(ka-specification grapher),` which instructs DASH to generate a graph editor, rather than a list browser. Figure 9 shows the resulting graph editor. The grapher widget set consists of a grapher canvas, a set of add buttons for new nodes, a delete button, and an edit button. The MEDITOR tool run-time system allows the target-tool user to edit graph nodes by double clicking on them. This oper-

Figure 7: A sample form generated by DASH. This form is based on the definition of the `corporation` class (Figure 6).

ation will open an editor for the instance that the graph node represents. In certain domains, domain experts must be able to edit properties of links (such as preconditions for transitions). In out approach, links among nodes can represent instances (if specified by the developer in the ontology). By double clicking on the links, the target-tool user can edit the link instances.

DASH can map slots of cardinality multiple (i.e., lists) to *table* selectors, and to *matrices* of widgets. Matrices, however, are not supported by CLIPS directly. The developer can indicate that a list of integers, for example, should be edited as a matrix in the target tool. DASH and MEDITOR assume that the upper left-hand corner of a matrix corresponds to the first element in the list. Moreover, DASH and MEDITOR take advantage of the size of the matrix (as specified by the developer) in mapping the list to the matrix, and vice versa. There are some restrictions on the allowed element data types for lists edited by matrices. Currently,

```
(defclass workflow (is-a USER)
     (slot name (type string))
     (slot flow-graph (type instance)
          (cardinality multiple)
          (allowed-classes personal-activity group-activity
                           external-activity composed-activity)
          (ka-specification grapher))
 )
```

Figure 8: The sample definition of the `workflow` class.

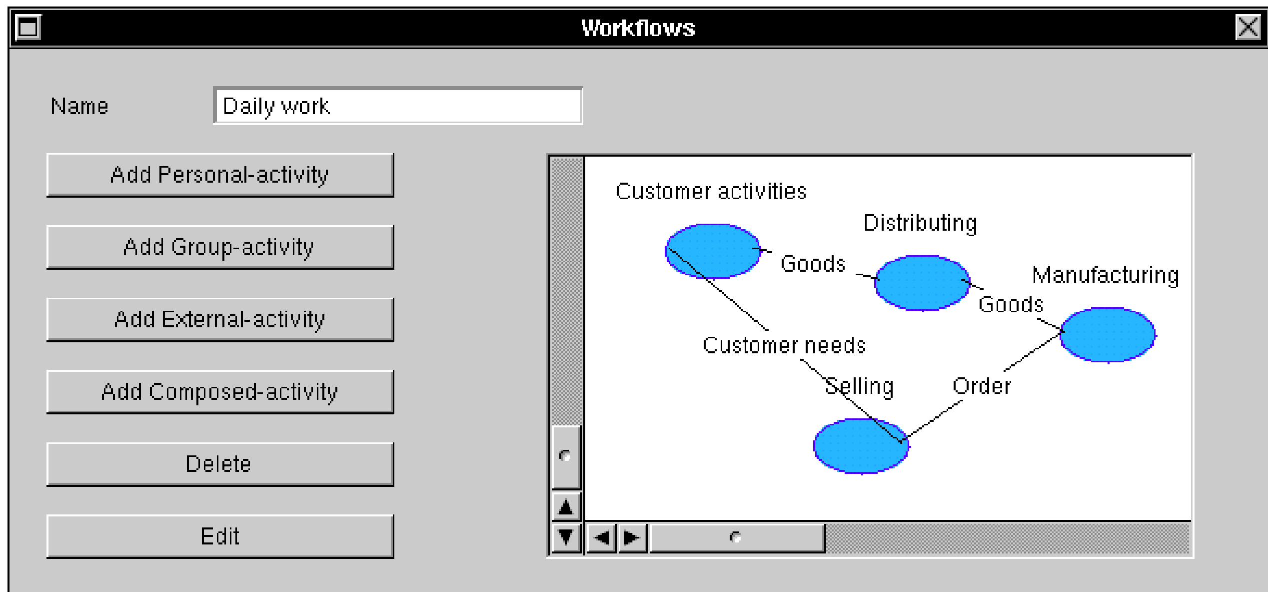

Figure 9: A sample form with a graph editor. The *add* buttons allow the tool user to add new nodes to the graph. The tool user can define relationships among the nodes by connecting them with links. Furthermore, the tool user can edit node and link definitions by selecting them and clicking on the *edit* button.

the element data types supported are integer, float, string, boolean, and symbol.

5.2 Layout Rules

When DASH has completed the instantiation of the widgets, it invokes the layout subsystem. This subsystem uses a relatively straightforward layout algorithm, because the layout is intended as a starting point for manual custom adjustments. The layout subsystem uses a set of rules for determining the initial size of the widgets, and the relative widget positioning for widget sets (e.g., list browsers). The layout algorithms then positions the widgets in rows and columns on the window, and aligns them to a grid. Finally, the layout algorithm determines the size of the window. In addition to layout of new windows, the layout subsystem supports *incremental layout* of widgets added to windows custom tailored previously (due to slots added to the ontology). The incremental-layout algorithm positions new widgets below the preexisting widgets, and resizes the window to accommodate the new widgets.

The hierarchical design approach and the separation of function (selectors) and implementation (widgets) make it easier to maintain DASH's design rule base. The DASH developer can add support for new data-entry devices in the target tool. For instance, the DASH developer can add pop-up menus for small integers by mapping the numeric-entry selector to a different widget implementation for small integers (e.g., for integer slots where the minimum value is zero and the maximum value is ten). In the next section, we shall discuss tool support for maintenance of DASH and the design rules.

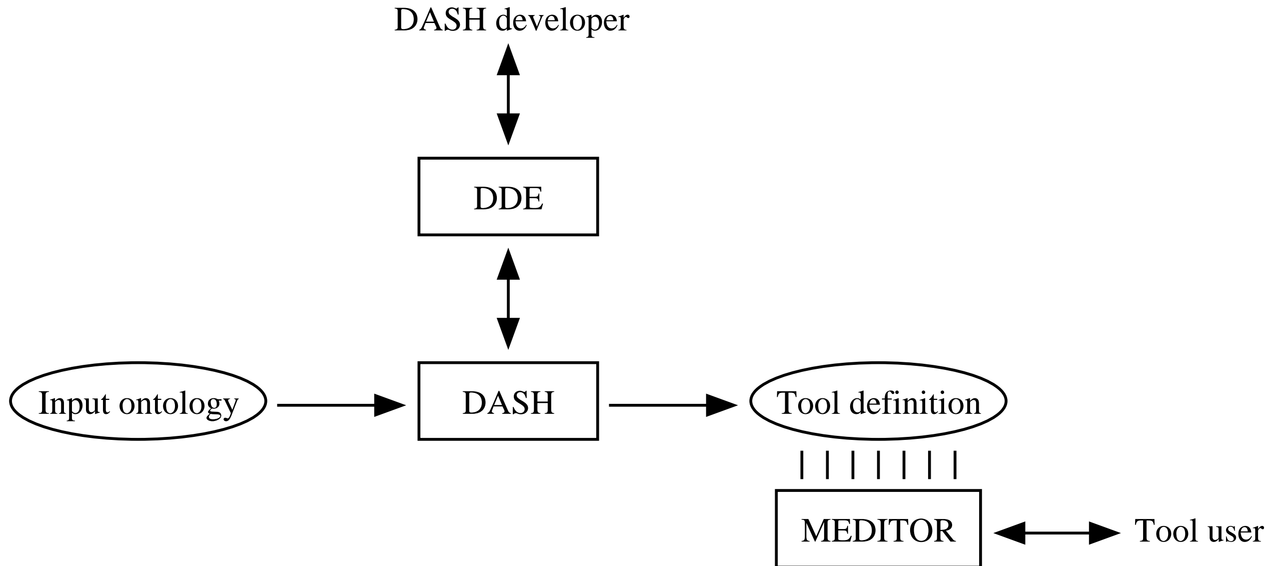

Figure 10: The relationships among DASH, DDE, and MEDITOR. DDE can be viewed as a meta-metatool for the DASH developer.

6 The DASH Development Environment

The DASH system and the design rules used by DASH evolved gradually over time, because of the increasing use of and the requirements on the target tools. At one point, the maintenance of existing DASH functionality required much of the resources available for DASH development. Thus, we found it difficult to extend DASH further, and to add new functionality. This situation warranted a new approach to the development and maintenance of DASH.

The DASH Development Environment (DDE) is a system that assists developers in the design, implementation, and maintenance of DASH (Figure 10). DDE allows the DASH developer to navigate and edit the DASH design-rule base, run diagnostic tests, and inspect graphically the intermediate design representations generated by DASH. Moreover, DDE supports the configuration of a *mapping method* that performs transformation among design stages in DASH. The mapping method uses a set of rules to map tentative tool-design representations to refined representations.

Figure 11 shows the structure of the mapping method. The method builds an *index* of the elements in the *input structure,* and performs an *abstraction* of key input features based on the input and index. The method uses information from the abstraction step to guide the generation of an intermediate representation, *templates,* from the input structure. The templates represent tentative elements for the output structure. The method then uses the templates as the basis for the generation of the *target structure,* which is the output of the method. The method generates target-structure elements by instantiating the templates. Sets of *mapping rules* define the mapping between the stages in the mapping method. DASH uses four instances of this mapping method as the basis for tool design. The first instantiation generates the dialog structure from the ontology. The second instantiation generates selectors from the ontology and from the dialog structure. The third instantiation maps the selectors to appropriate widgets, and the fourth instantiation generates the final output from the widgets.

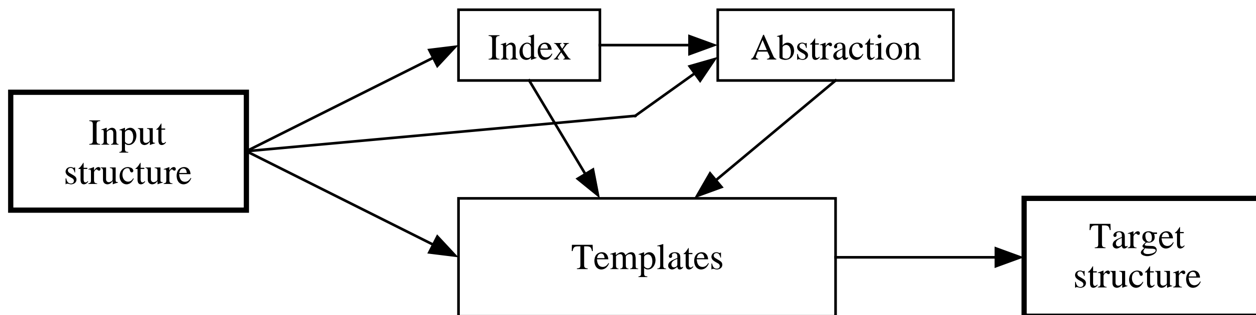

Figure 11: The structure of the mapping method modeled by a data-flow graph. The method indexes the input structure, and abstracts key features of the input structure. The method then uses the index and abstraction information to perform the mapping from the input structure to templates (for the target structure). Finally, the method instantiates the templates to produce the target structure, which is the output of the method.

DDE allows the developer to edit the mapping definition and to navigate rule sets and other definitions. The DDE user interface provides a main menu that illustrates the data-flow relationships in DASH. The user can access mapping definitions, ontologies, and subsystems of DASH. The graphical navigation facilities of DDE helps the developer to locate rapidly the code segment responsible for a certain functionality in the system.

DDE's subsystem for diagnostic tests allows the developer to run test cases up to certain points in the generation process of DASH. These test cases consist of input ontologies designed to test various tool-design rules in DASH. DDE includes a set of common test cases, and the developer can specify additional ontologies as test cases. The latter feature is useful for debugging DASH problems that occur with specific input ontologies (perhaps provided by DASH users).

After a test is completed, DDE will provide an interactive test protocol that the DDE user can use to analyze the result of the test run (Figure 12). The interactive protocol provides graphical inspectors for intermediate design representations generated by DASH. The DDE user can inspect these structures at different levels of granularity to verify that the mapping steps are working correctly. By running problematic ontologies as test cases in DDE, the DASH developer can inspect the intermediate results of the generation process, and can isolate rapidly any problem in the mapping steps of DASH.

We designed DDE as an application-specific development environment for DASH (analogous to domain-oriented knowledge-acquisition tools). The DDE user interface incorporates concepts relevant to the DASH architecture and for the instantiations of the mapping method. DDE has proven to be helpful in the development and maintenance of DASH, and has allowed us to extend DASH with fewer resources. The number of bugs in DASH has decreased, and the time required for debugging has been reduced significantly.

In principle, the class of application-specific tools that DDE represents can be used to support the development and maintenance of other types of software. Potentially, metalevel systems, such as PROTÉGÉ-II, can be used to assist the development of application-specific tools. Although the current version of PROTÉGÉ-II cannot generate tools such as DDE, because

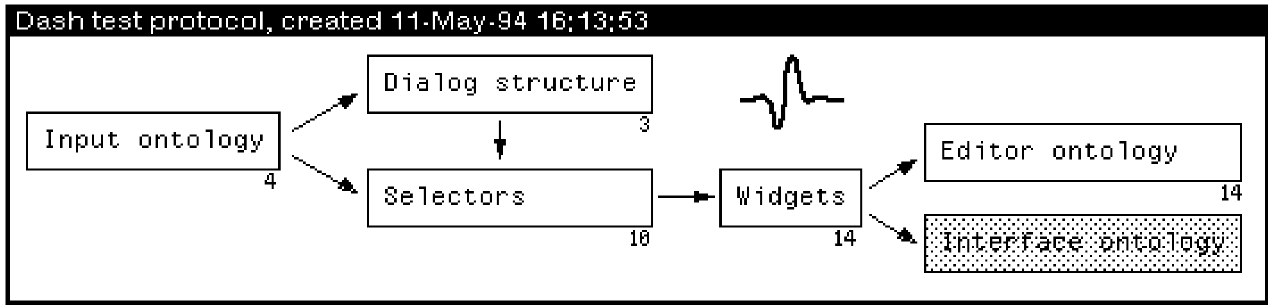

Figure 12: Interactive test protocol generated by DDE. This panel allows the DDE user to inspect intermediate design representations, such as the dialog, selector, and widget structures, produced by DASH in the test run.

the DASH tool-design rules and the MEDITOR tool run-time system are designed for knowledge-acquisition tools, a similar approach and architecture can be created for other classes of custom-tailored tools.

7 Tools Generated

By using application ontologies as the basis for generation of knowledge-acquisition tools, the developer can specify knowledge-acquisition tools for a wide variety of domains. To illustrate the design space of the target tools, we shall mention briefly some of the knowledge-acquisition tools developed with PROTÉGÉ-II.

One of the first application domains for which we generated knowledge-acquisition tools was the Sisyphus room-assignment problem [9]. In this example problem, the task is to assign office workers to rooms in an office building under certain constraints (e.g., room size and location in accordance with the workers job description). The Sisyphus room-assignment problem is a standard problem used by researchers in knowledge acquisition to compare different approaches to knowledge acquisition, knowledge representation, and problem solving.

The application ontology for the room-assignment task is based on the concepts *person, room,* and *professional role.* Only the professional-role class is relevant for knowledge acquisition, because persons and rooms are run-time input data to the system. From the perspective of knowledge-acquisition-tool support, the room-assignment task is relatively simple. A knowledge-acquisition tool consisting of only a few forms is sufficient to create the knowledge base. A browser of professional roles provides access to a form for specification of room requirements for each professional role.

To test PROTÉGÉ-II on a more difficult task than the room-assignment problem, we developed a system that assists users in choosing appropriate equipment to rent when moving (e.g., trucks and trailers). This example is more complex than the room-assignment problem, but it is less complex than a realistic task for a knowledge-based system. The knowledge-acquisition tool for this system acquires specifications for rental equipment (e.g., trucks, trailers, and boxes), variables that are involved in the configuration process (e.g., weight and cost), and constraint rules and fixes. The application ontology for this system consists of 12 classes, and the knowledge-acquisition tool generated consists of 11 forms.

The Vertical Transportation (VT) task is a standard problem based on an existing knowledge-based system for elevator configuration [11]. Analogous to the room-assignment task, the Sisyphus VT problem is a well-documented task that is used by researchers to compare knowledge-acquisition and problem-solving approaches. The VT task, however, represents a significant problem that requires a knowledge-based system of significant size and complexity. The implementation of the original VT system was supported by SALT [10], which is a method-specific knowledge-acquisition tool that acquires knowledge for the propose-and-revise method. We use the same problem-solving method (propose and revise) to perform both the rental-equipment task and the VT task [19]. Although the method ontology is the same, the domain and application ontologies (and, thus, the knowledge-acquisition tools) differ. The application ontology for the VT system consists of 78 classes, and the knowledge-acquisition tool generated consists of 38 forms. The resulting knowledge base consists of 674 instances of concepts in the application ontology.

T-HELPER [15] is a system that assists physicians by providing computer-based support for protocol-directed therapy. The T-HELPER system is based on a set of protocol definitions that represent the context in which the protocol can be used, and define the algorithmic procedure (skeletal plan) of the protocol. The T-HELPER knowledge-acquisition tool allows the expert physician to enter information about the protocol in forms and to define the algorithm graphically [20]. The current version of the application ontology used for tool generation consists of 76 classes, and the knowledge-acquisition tool generated consists of 25 forms. The T-HELPER knowledge base is currently under construction in our laboratory.

The knowledge-acquisition tools for the room-assignment and rental-equipment tasks are relatively simple. However, when the complexity of the knowledge-base design increases, the advantage of generating tools automatically increases as the burden of implementing and maintaining such tools manually becomes unmanageable. The knowledge-acquisition tools for the VT and T-HELPER applications show how generated tools assist knowledge acquisition for relatively complex tasks.

8 Summary

Domain-specific knowledge-acquisition tools enable domain specialists to create knowledge bases with minimal assistance from system developers. This tool support is often essential to the initial success and subsequent maintainability of knowledge-based systems, because of the complexity and size of the knowledge bases. Developers can custom tailor domain-specific tools for the requirements of individual users, and of development projects by using a metatool to generate knowledge-acquisition tools automatically.

The PROTÉGÉ-II system shows that the generation of knowledge-acquisition tools from application ontologies is a valid approach. However, in addition to the metatool DASH, the developer must have access to tools for editing ontologies, and to a run-time system for the knowledge-acquisition tools generated. If the developers reuse a preexisting problem-solving method, the output of the target knowledge-acquisition tool must be mapped to the format used by the method. Compared to its predecessors, PROTÉGÉ-II provides a solid platform for the generation of domain-specific knowledge-acquisition tools. We have used PROTÉGÉ-II successfully for knowledge acquisition in several projects, and the PROTÉGÉ-II suite of tools has provided significant support and reduced the development time considerably.

We plan to extend PROTÉGÉ-II by providing graphical library functions for reusable

problem-solving methods, by improving the support for mappings among ontologies, and by providing additional design support for user interfaces. So far, we have concentrated our research on tools for the development of knowledge-based systems. However, many of the principles for tool generation apply to the development of other types of software as well. In our continued research, we plan to generalize results from knowledge-acquisition tools for knowledge-based systems, and to explore how metatools can support the software-development process by automating the generation of project-specific tools.

Acknowledgements

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 gifts from Digital Equipment Corporation and from the Computer-Based Assessment Project of the American Board of Family Practice. Dr. Musen is recipient of National Science Foundation Young Investigator Award IRI-9257578.

We thank John W. Egar, Yuval Shahar, and Eckart Walther for valuable discussions and suggestions on generation of knowledge-acquisition tools in PROTÉGÉ-II. On-line information about PROTÉGÉ-II and DASH is available through a World-Wide-Web (WWW) service (`http://camis.stanford.edu/protege/`).

References

[1] Henrik Eriksson. Metatool support for custom-tailored domain-oriented knowledge acquisition. *Knowledge Acquisition*, 4(4):445–476, 1992.

[2] Henrik Eriksson and Mark A. Musen. Metatools for software development and knowledge acquisition. *IEEE Software*, 10(3):23–29, 1993.

[3] Henrik Eriksson, Angel R. Puerta, and Mark A. Musen. Generation of knowledge-acquisition tools from domain ontologies. *International Journal of Human–Computer Studies*, in press.

[4] Henrik Eriksson, Yuval Shahar, Samson W. Tu, Angel R. Puerta, and Mark A. Musen. Task modeling with reusable problem-solving methods. *Artificial Intelligence*, in press.

[5] John H. Gennari, Samson W. Tu, Thomas E. Rothenfluh, and Mark A. Musen. Mapping domains to methods in support of reuse. *International Journal of Human–Computer Studies*, in press.

[6] Jeff Johnson. Selectors: Going beyond user-interface widgets. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '92)*, pages 273–279, Monterey, CA, May 3–7 1992. ACM, New York.

[7] Werner Karbach, Marc Linster, and Angi Voß. Models, methods, roles and tasks: Many labels—one idea? *Knowledge Acquisition*, 2(4):279–299, 1990.

[8] Atsuo Kawaguchi, Hiroshi Motoda, and Riichiro Mizoguchi. Interview-based knowledge acquisition using dynamic analysis. *IEEE Expert*, 6(5):47–60, October 1991.

[9] Marc Linster, editor. *Sisyphus'92: Models of Problem Solving*, Technical Report 630, Gesellschaft für Mathematik und Datenverarbeitung (GMD), St. Augustin, Germany, 1992.

[10] Sandra Marcus and John McDermott. SALT: A knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence*, 39(1):1–37, 1989.

[11] Sandra Marcus, Jeffrey Stout, and John McDermott. VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, 9(1):95–112, Spring 1988.

[12] David Marques, Geoffroy Dallemange, Georg Klinker, John McDermott, and David Tung. Easy programming: Empowering people to build their own applications. *IEEE Expert*, 7(3):16–29, June 1992.

[13] John McDermott. Preliminary steps toward a taxonomy of problem-solving methods. In Sandra Marcus, editor, *Automating Knowledge Acquisition for Expert Systems*, chapter 8, pages 225–256. Kluwer Academic Publishers, Boston, MA, 1988.

[14] Mark A. Musen. Automated support for building and extending expert models. *Machine Learning*, 4:349–377, 1989.

[15] Mark A. Musen, Robert W. Carlson, Lawrence M. Fagan, Stanley C. Deresinski, and Edward H. Shortliffe. T-HELPER: Automated support for community-based clinical research. In *Proceedings of the Sixteenth Annual Symposium on Computer Applications in Medical Care*, pages 719–723, Washington, D.C., November 1992.

[16] NASA. *CLIPS Reference Manual.* Software Technology Branch, Lyndon B. Johnson Space Center, NASA, Houston, TX, 1991.

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

[18] Angel R. Puerta, John W. Egar, Samson W. Tu, and Mark A. Musen. A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. *Knowledge Acquisition*, 4(2):171–196, 1992.

[19] Thomas E. Rothenfluh, John H. Gennari, Henrik Eriksson, Angel R. Puerta, Samson W. Tu, and Mark A. Musen. Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTÉGÉ-II solutions to Sisyphus-2. In *Proceedings of the Eighth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 43.1–43.30, Banff, Canada, January 1994.

[20] Samson W. Tu, Henrik Eriksson, John H. Gennari, Yuval Shahar, and Mark A. Musen. Ontology-based configuration of problem-solving methods and generation of knowledge-acquisition tools: Application of PROTÉGÉ-II to protocol-based decision support. *Artificial Intelligence in Medicine*, in press.