Generation of Knowledge-Acquisition Tools from Domain Ontologies

Henrik Eriksson Angel R. Puerta Mark A. Musen

Medical Computer Science Group
Knowledge Systems Laboratory
Stanford University School of Medicine
Stanford, California 94305-5479, U.S.A.

Abstract

Metalevel tools can support the software development process by automating the design of task- and application-specific tools. DASH is a metalevel tool that allows developers to generate domain-specific knowledge-acquisition tools from domain ontologies. Domain specialists use the knowledge-acquisition tools generated by DASH to instantiate the concepts and relationships defined in the domain ontologies. The output of the knowledge-acquisition tools is a collection of instances that constitute the knowledge base for a knowledge-based system.

To automate the generation of appropriate tools, the DASH architecture uses a dialog-design module to produce a dialog structure that defines the target tool at the editor and window level. Given the dialog structure, a layout-design module completes the window layouts. DASH allows the developer to custom tailor the layout of the knowledge-acquisition tool for its users, and to store such modifications persistently so that they can be reapplied when the target tool is regenerated. The DASH implementation is based on a mapping problem-solving method that defines the tool-design steps. The Dash Development Environment (DDE) is an application-specific environment that supports the configuration of the mapping method and the maintenance of DASH. We have used DASH to generate several knowledge-acquisition tools for a broad range of application tasks.

1 Tool Generation

Much of the current research on knowledge acquisition from domain experts is focused on the reuse of predefined *ontologies* and *problem-solving methods* for knowledge-based systems, and on tools that support knowledge acquisition for such architectures (Karbach, Linster & Voß, 1990; Marques, Dallemange, Klinker, McDermott & Tung, 1992; McDermott, 1988; Neches, Fikes, Finin, Gruber, Senator & Swartout, 1991; Steels, 1990; Puerta, Egar, Tu & Musen, 1992). An ontology is a declarative model of the terms and relationships in a domain. Typically, ontologies are organized as class hierarchies, where each class defines a set of objects of a certain type (Neches *et al.*, 1991). Each class has a set of attributes, often

called slots, which models the concept that the class represents. A problem-solving method is a domain-independent strategy for performing a task, such as classification, design, or planning. Such problem-solving methods take advantage of knowledge bases in performing their task. Method-specific knowledge-acquisition tools can support knowledge acquisition for problem-solving methods. Based on their methods' problem-solving model, such tools acquire from domain specialists the knowledge required to accomplish the task. However, because reusable methods use concepts defined primarily by computer scientists, nonprogrammers find it difficult to use knowledge-acquisition tools that are based on such conceptual models (Musen, 1989*a*).

Domain-specific tools, on the other hand, are based on the terms and relationships of the application domain. Domain specialists interact with such tools by using their own terminology. OPAL (Musen, Fagan, Combs & Shortliffe, 1987) is an example of a domain-specific knowledge-acquisition tool that is based on a strong domain model—treatment plans for cancer. Domain experts use OPAL to edit graphically skeletal plans for the ONCOCIN advisory system (Tu, Kahn, Musen, Ferguson, Shortliffe & Fagan, 1989). The disadvantage of domain-specific tools is that they require much work for implementation and maintenance. Researchers are solving this problem with *metatools* that generate domain-specific tools automatically from high-level specifications (Eriksson & Musen, 1993).

The metatool PROTÉGÉ-I (Musen, 1989*a*; Musen, 1989*b*) uses instantiation of a problem-solving method for an application domain as the basis for the generation of domain-specific knowledge-acquisition tools. PROTÉGÉ-I demonstrated how metatools can instantiate automatically tools similar to OPAL for new domains. The drawback of traditional method-specific approaches, however, is that the metatool is coupled to a specific problem-solving method (e.g., skeletal planning) for the application system, and that the metatool cannot generate target tools for other problem-solving methods. The metatool Spark (Marques *et al.*, 1992) uses a generalized approach in that it associates a method-specific, configurable knowledge-acquisition tool with each problem-solving method in a library of predefined methods. In this approach, knowledge acquisition for systems constructed from several problem-solving methods require a set of such knowledge-acquisition tools. Although the Spark approach addresses the problem of multiple and alternative problem-solving methods, it is difficult to ensure that a collection of primitive knowledge-acquisition tools provides a coherent view to the domain specialists.

The metatool DOTS (Eriksson, 1992; Eriksson, 1993) uses an architectural specification of the target knowledge-acquisition tool as the basis for the tool generation. Although DOTS is general with respect to the target tools that it can generate, we have found that DOTS is difficult to use for developers who are unfamiliar with the design and generation of domain-specific knowledge-acquisition tools. DOTS solves much of the software-engineering problem of implementing a domain-specific knowledge-acquisition tool, but it provides little support for the design of an appropriate knowledge-acquisition tool for an application domain.

In this paper, we describe DASH, a metatool that generates knowledge-acquisition tools from domain ontologies. DASH designs a knowledge-acquisition tool by first generating a *dialog structure*—an abstract map of the editor and window structure of the tool, and of how the user might proceed among the windows. Next, DASH generates the details of each window based on the dialog structure, and on the class definitions in the input ontology. When DASH has completed the window layouts, the metatool allows the developer to custom tailor the target tool for an application domain, and for individual users (using a graphical interface-

design tool). The output of DASH is a declarative specification of the target tool that can be executed by a run-time system, or, alternatively, compiled into C code.

DASH is part of the PROTÉGÉ-II environment, which supports development of knowledge-based systems from reusable problem-solving methods (Puerta *et al.*, 1992; Puerta, Tu & Musen, 1993). Also, the PROTÉGÉ-II architecture includes support for defining and editing domain ontologies. Because DASH is a relatively complex system to maintain and extend, we have designed the Dash Development Environment (DDE), which is a graphical tool that supports the maintenance of DASH.

This article is organized as follows: Section 2 introduces the use of metatools in application development. Section 3 discusses the use of the domain ontologies as the basis for tool generation. Section 4 presents the DASH architecture. Section 5 provides an illustrative example of tool generation from domain ontologies in DASH. Section 6 discusses DDE and its use. Section 7 summarizes and discusses the DASH approach; conclusions are drawn in Section 8.

2 Development with Metalevel Tools

Knowledge acquisition and system development with metalevel tools differ from conventional software development in several aspects, because the development process involves tool design. We are interested primarily in knowledge-acquisition tools that allow nonprogrammers to edit knowledge structures graphically. Such knowledge editors must incorporate a domain model that is valid cognitively for the tool users. Before discussing how we can adapt development methodologies to take advantage of metatools, we must first establish the role of the target tools in the development process.

We shall exemplify the type of knowledge-acquisition tools that we are generating by examining parts of a tool generated by DASH. Figure 1 shows a domain-specific form from a knowledge-acquisition tool that allows physicians to edit treatment plans for clinical trials (clinical-trial protocols). This form allows oncology specialists to define the relevant aspects of the treatment plan in domain-specific terms. The tool user may enter the information in any order. In Figure 1, the tool user has entered "Pyrimethamine Therapy for Prevention of Toxoplasma" as the protocol title in the form. Moreover, the tool user has provided a list of the investigators responsible for the protocol design. By selecting one of the items in the investigator browser, and by clicking on the edit button, the user can edit information—in a separate form—about each investigator (such as institution and telephone number). Note that the form incorporates domain terms that are meaningful to domain specialists, but not necessarily to other people. This knowledge-acquisition tool is similar to OPAL (Musen *et al.*, 1987) in that it acquires knowledge for management of treatment plans. The principal difference between these tools is that OPAL is specific to the cancer-treatment domain, whereas the tool shown in Figure 1 is specific to AIDS treatment.

The use of knowledge-acquisition tools affects the development methodology in that domain specialists participate actively in the development process by providing directly much of the domain knowledge required. In other words, the domain specialists perform a detailed system design by providing the knowledge structures that defines the knowledge base. The role of a metatool in this development approach is to enable the developer to provide a suitable tool for the domain specialists.

Many conventional development methodologies can be adapted to take advantage of meta-

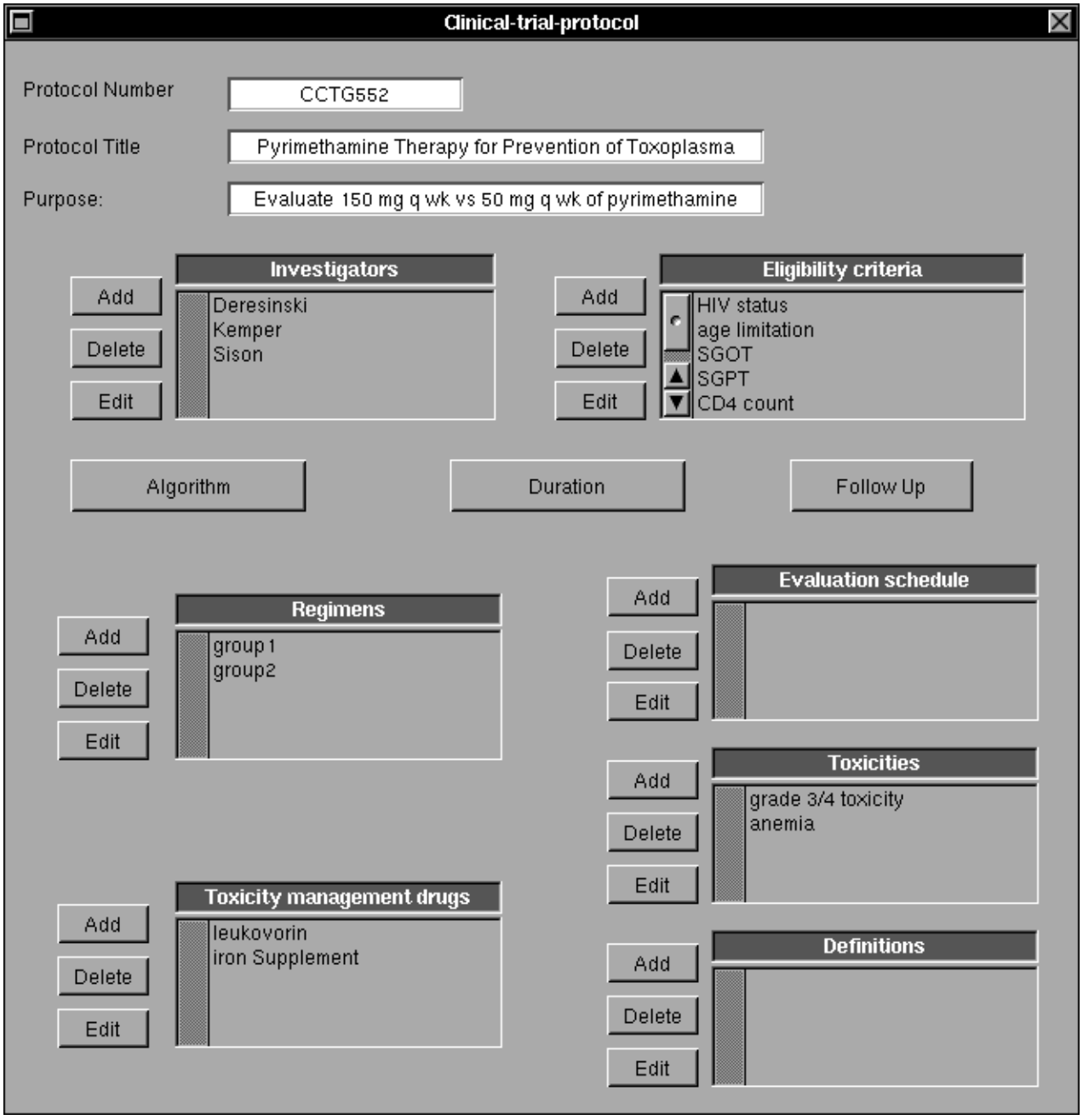


FIGURE 1. Domain-specific form for the definition of clinical-trial protocols for AIDS treatment generated by DASH. At the top of this form, there are fields for the protocol number, title, and purpose. Moreover, this form provides a collection of browsers that allows the tool user to add, delete, and edit definitions for investigators, eligibility criteria, and so on. The buttons labeled "Algorithm," "Duration," and "Follow Up" provide access to subforms that domain specialists use to define these aspects of the protocol.

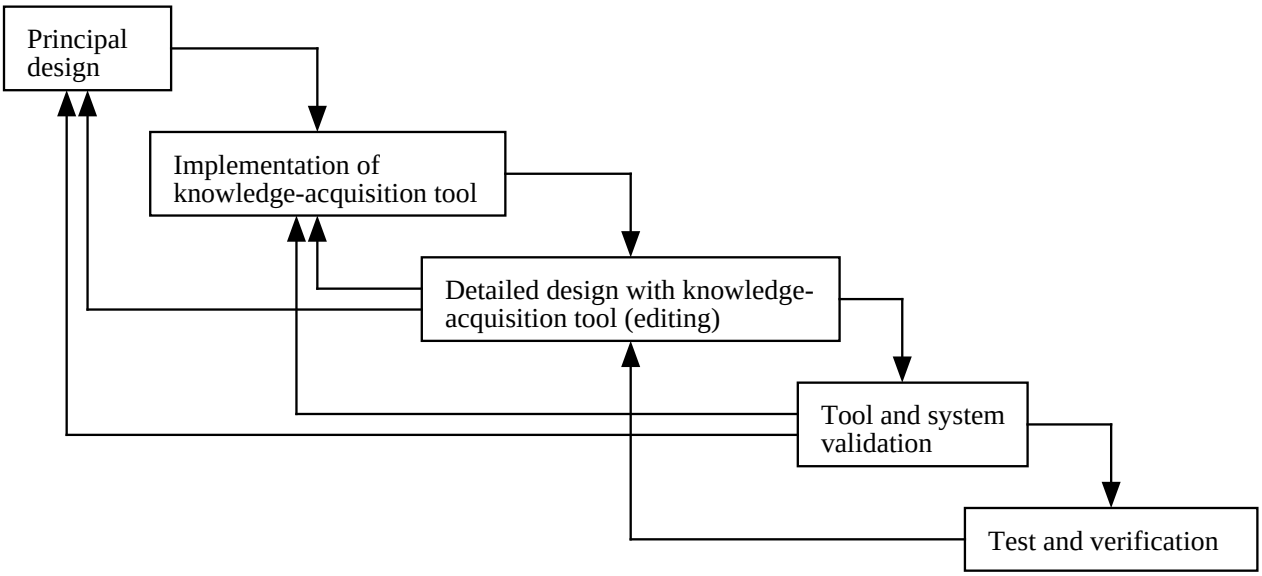


FIGURE 2. Development methodology with a metatool. The developer completes the principal design of the target system (e.g., a domain ontology), and implements a knowledge-acquisition tool, which is used by domain specialists to edit domain knowledge (detailed design). After using the tool for some time, the developer and the tool user validate the tool and its output. Note that validation of the knowledge-acquisition tool and the system might induce changes to the principal design, such as changes to a domain ontology. Finally, the developer submits the application system to test and verification. Although this approach is based on the waterfall model, it is possible to take advantage of metatools in other methodologies, such as iterative and spiral models.

tools. Figure 2 shows a development model that incorporates the use of metatools and knowledge-acquisition tools. (Note that this model is based on the waterfall model. Compared to the strict waterfall model, we have relaxed the constraints on iteration in our model.) Here, the developer analyzes the application task, and makes a principal design of the system. In the DASH approach, the principal design consists of a domain ontology that can be used by DASH to generate the target tool. The developer may want to revise the principal design after testing the knowledge-acquisition tool. The extended use of a knowledge-acquisition tool by the domain specialists can uncover shortcomings in the tool, and in the principal design. In these cases, the developer must revise the principal design, and must regenerate the knowledge-acquisition tool. Typically, the developer maintains the knowledge-acquisition tool over the application's life cycle.

3 Ontologies as the Basis for Knowledge-Acquisition Tools

The knowledge-acquisition tools that we generate from domain ontologies are designed to edit instances of the ontologies by providing a collection of domain-specific editors for the

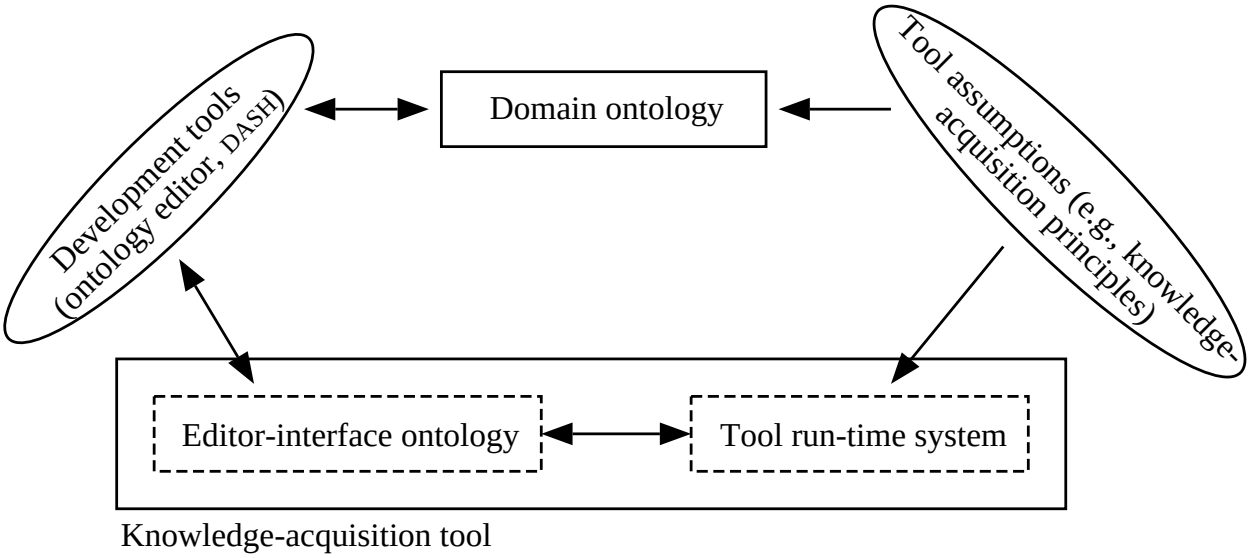


FIGURE 3. Design of knowledge-acquisition tools from domain ontologies. This model is inspired by model-based design of user interfaces (de Baar *et al.*, 1992). A high-level application model, in our case a domain ontology, is the basis for the tool design. The instances of the editor-interface ontology together with the tool run-time system form the target knowledge-acquisition tool. PROTÉGÉ-II provides a set of development tools for editing the domain ontology, and for generating the editor-interface ontology. Note that model-based design must make assumptions about the target system. DASH, for instance, makes the assumption that the target tools are domain-specific knowledge-acquisition tools.

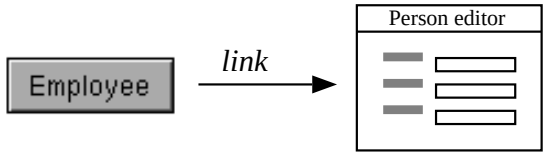
tool user. For example, when domain specialists fill out forms in the tools, they are editing instances of the classes that defined the forms. The tool users can save the information entered in these editors persistently for later use, and can produce instance definitions suitable for the application system from the information in these editors. The application system uses both the classes in the ontology and the instances of these classes provided by the knowledge-acquisition tool.

The DASH approach to the generation of knowledge-acquisition tools is inspired by *model-based design* (Figure 3). Model-based design is an approach where an application model is used as the basis for the design of the target system. Researchers in human–computer interaction are using application models as the basis for the user-interface design (de Baar, Foley & Mullet, 1992). In the DASH approach, the target system is the knowledge-acquisition tool. We are using the domain ontology as the model for the tool application, and are using DASH to generate instances of the editor-interface ontology, which define the target tool, from the domain ontology.

The basic idea for tool generation from ontologies is that a metatool can map the data types of slots in a class definition to user-interface components of the knowledge-acquisition tool. Typically, the metatool generates editors for instances of each class in the ontology based on such mappings. The target knowledge-acquisition tool consists of a collection of such editors. Although the basic idea is relatively simple, there are many fundamental and technical problems that must be addressed before we can generate knowledge-acquisition tools

```
(slot chapter11 (type boolean))
```


```
(slot employee (type instance)
  (allowed-classes person))
```



(a) (b)

FIGURE 4. Example of a simple mapping from slot definitions to user-interface components. (a) Definitions for the Boolean slot `chapter11` and for the instance-pointer slot `employee`. (b) Form fields (widgets) for these slots. The user edits the value of the slot `chapter11` by clicking on a check box. Because the slot `employee` is a pointer to another instance, the slot is edited by clicking on the employee button that opens a "person" editor.

that are useful for practical software development. For instance, the metatool must analyze the ontology, and must create a high-level design of the target tool before it can apply such slot–interface mappings, because the metatool must establish the editors before it can design the editors' user interfaces. Moreover, the metatool must identify the slots that define the relationships among instances of different classes to design the connections among different editors in the knowledge-acquisition tool.

Let us examine an example of how a metatool can use class definitions in the ontology to generate form fields in a knowledge-acquisition tool. Figure 4 shows an example a simple mapping from slots in a class definition to user-interface widgets. The major advantage using domain ontologies as the basis for tool generation is that such ontologies, at least in object-oriented approaches, are defined as part of the regular development process. Thus, both the knowledge-acquisition tool and the final application system are based on the same ontology. Also, this approach combines the level of flexibility provided by DOTS with the level of design support provided by PROTÉGÉ-I. In the DASH approach, the additional effort required for developing a knowledge-acquisition tool for the application domain is small, which makes use of domain-specific tools cost effective.

4 The DASH Architecture

The metatool DASH generates knowledge-acquisition tools from domain ontologies. Typically, the developer uses the ontology editor provided by PROTÉGÉ-II to define such ontologies. The developer uses DASH to generate an initial version of the tool, which can be custom tailored by the developer for the domain and for the tool user.

The DASH architecture consists of a *dialog-designer* module and a *layout-designer* module. DASH designs the target tools top down by first invoking the dialog designer, and then running the layout designer. The dialog designer creates an abstract description of the editors in the knowledge-acquisition tool and their relationships. The layout designer instantiates the

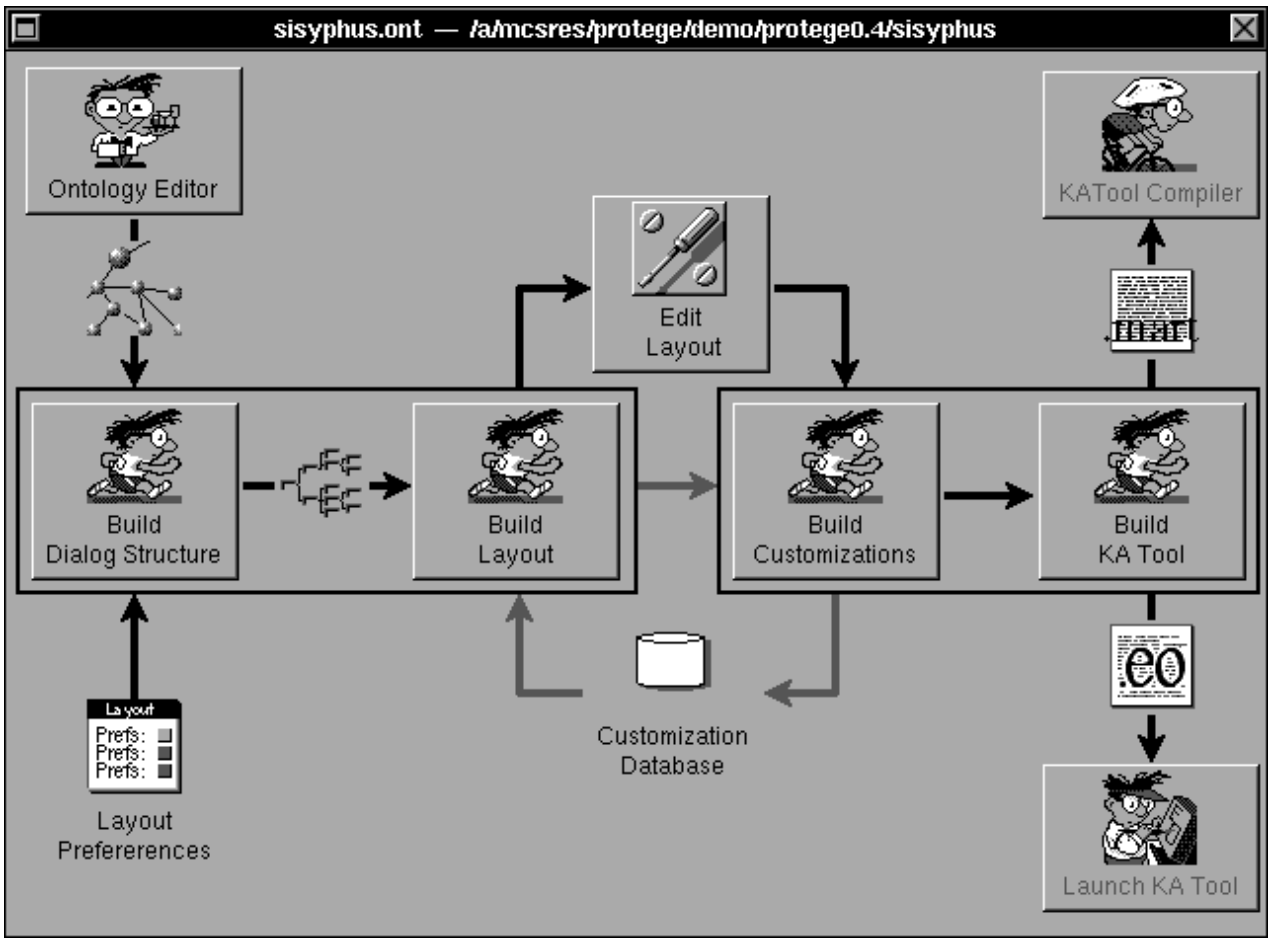


FIGURE 5. The dashboard. This control panel shows the main components of DASH, and allows the developer to control the generation of knowledge-acquisition tools.

editors—in particular, their window layout. Figure 5 shows the control panel for the DASH system. Developers use this panel to invoke tool generation in DASH, to inspect intermediate results, and to custom tailor the knowledge-acquisition tool. We shall discuss the most important subcomponents of the DASH architecture.

4.1 Dialog Design

The generation of knowledge-acquisition tools from domain ontologies is not merely a matter of mapping slot data types to user-interface components. Before a metatool can apply this mapping, it must design the overall structure of the knowledge-acquisition tool. The purpose of the dialog-designer module is to establish this high-level structure of the target tool. We use the term *dialog structure* to denote the configuration of windows, browsers, editors, and forms, and the way in which the tool user navigates these tool components. Such accessibility relationships are an important part of the design of knowledge-editing tools. In DASH, the dialog structure is represented as a graph where the nodes represent editors, and the links among the nodes represent accessibility relations. Thus, the dialog structure provides an

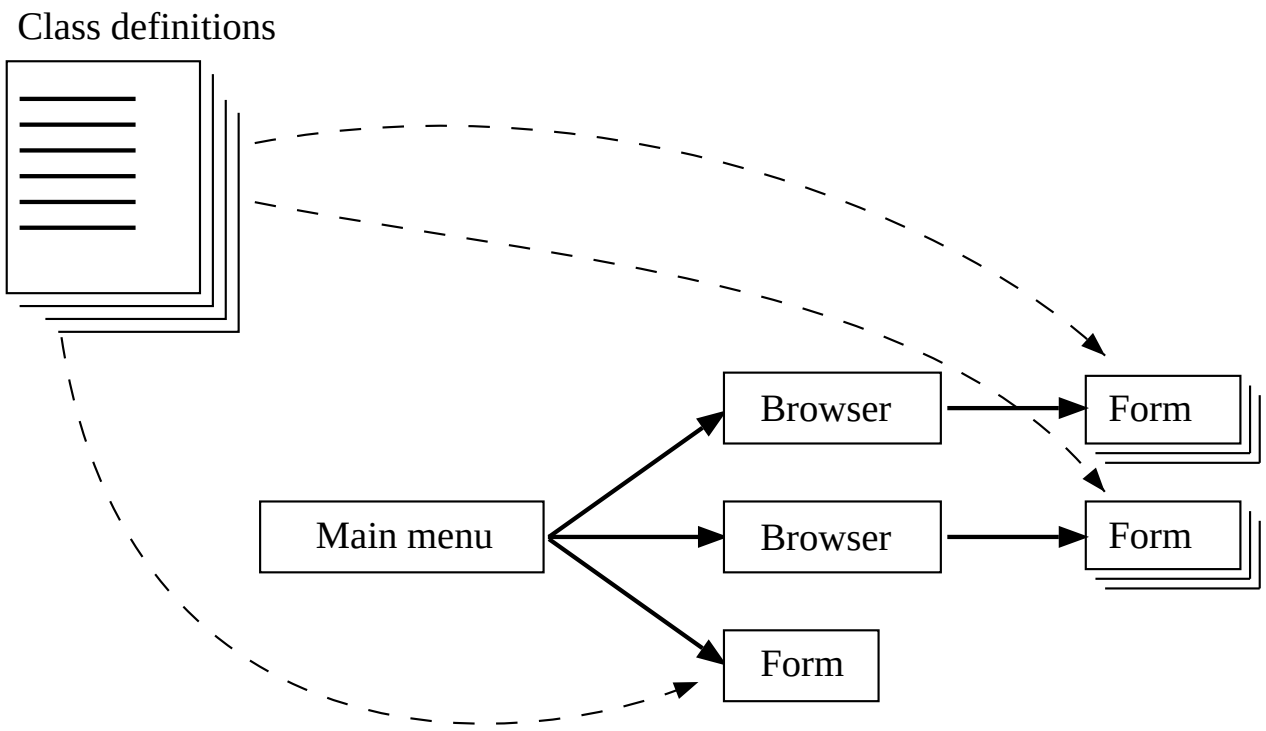


FIGURE 6. The mapping in DASH from class definitions in the input ontology to the dialog structure of the target tool. Most classes in the ontology correspond to a form node in the dialog-structure graph. DASH's dialog-designer module adds the browsers required to access multiple forms, as well as the main menu for the knowledge-acquisition tool. The main menu and the browsers are located on separate windows.

outline of the tool at the editor and window level.

DASH uses the relationships among classes in the domain ontology as the basis for the dialog structure. The dialog designer uses a set of design rules to produce the dialog structure automatically. First, the dialog designer analyzes the class definitions in the ontology, and builds an index of the class relationships. Next, the dialog designer traverses the class definitions, and creates editor templates for each of the relevant classes. As part of this process, the dialog designer links the editor templates according to the relevant class relationships (e.g., the *is-part* relations). Finally, the dialog designer adds the components required to access the editors, such as the main menu and scrollable list browsers for collections of items.

Figure 6 illustrates the generation of dialog structures from class definitions. The example dialog structure consists of the main menu of the target tool, which provides access to two browsers, and to a single form-based editor. In turn, these browsers provide access to lists of items that can be edited by other forms. The dashed arrows in Figure 6 illustrate how the dialog designer maps class definitions to form-based editors in the dialog structure. The dialog designer adds the main menu and the browsers to the graph automatically. Each node in the dialog structure, including the main menu and the browsers, corresponds to a screen window in the target tool.

The dialog designer uses information from the domain ontology to produce the dialog structure. The developer can declare the classes whose instances should be edited at the top

level of the knowledge-acquisition tool (i.e., editors, such as forms, for these instances should be available directly under the main menu). Typically, these classes represent the most important concepts in the domain. Other relevant classes may be available as subeditors to the top-level editors. Depending on the cardinality of the instances that the forms are editing, the dialog designer inserts appropriate browsers in the dialog-structure graph. The dialog designer inserts browsers if the corresponding class is declared to have more than one instance.

4.2 Layout Design

To complete a knowledge-acquisition tool, we must design the details of each editor and window in the dialog structure. The task of the layout-designer module is to instantiate the editors of the target tool given the dialog structure. The layout designer traverses the dialog-structure graph top down, and generates editor definitions based on the nodes in the dialog structure, and on the corresponding class definitions in the domain ontology.

The layout designer uses an intermediate representation of the editors. The first step in the layout-design process is to map the class slots to *selectors* (Johnson, 1992). Selectors are high-level representations of user-interface widgets. Selectors represent a function (e.g., selection of an item from a list), rather than the widget implementation of this function (e.g., pull-down menus and radio buttons). The next step is to instantiate the selectors to widgets, and to lay out the widgets on the forms and windows.

The layout-designer module is responsible for the window layouts. The layout designer uses a layout algorithm that allows the developer to set primary layout parameters, and to custom tailor the layout generated. The goal of the layout designer is not to generate the final layout, but rather to produce a layout that the developer can custom tailor easily. There are many fundamental difficulties in generating automatically appropriate layouts from ontologies, because the slot definitions contain insufficient and inconclusive information for layout purposes. Note that the layout algorithm operates on widgets rather than on selectors, because selectors do not have a definite geometry before they are instantiated to widgets. Also, the developer can custom tailor the widgets, but not the selectors. This approach allows the developer to adjust the details for the target tool's user interface (such as rearranging the items in a pull-down menu).

Figure 7 illustrates the mapping from the slots in a class definition to the selector representation, and to the widget representation. The layout designer uses a set of design rules to map each slot to a selector. This mapping is based primarily on slot data types. Another set of design rules defines the mapping between the selectors and their widget counterparts. The latter mapping is one to many, because several widgets might be required to implement certain selectors. For instance, a list browser selector is typically implemented by a collection of widgets, such as add, delete, and edit buttons, and the browser widget itself.

When the layout designer has instantiated and laid out the widgets, the developer may make custom adjustments to the layout (Figure 7). DASH allows the developer to custom tailor the user interface of the knowledge-acquisition tool using NeXT's Interface Builder (NeXT, 1990). Interface Builder is a graphical development tool for user interfaces. In this step, the developer can change the layout, and can relabel items by direct manipulation (see Figure 8). When the developer has completed the custom adjustments, DASH finalizes the specification of the target knowledge-acquisition tool.

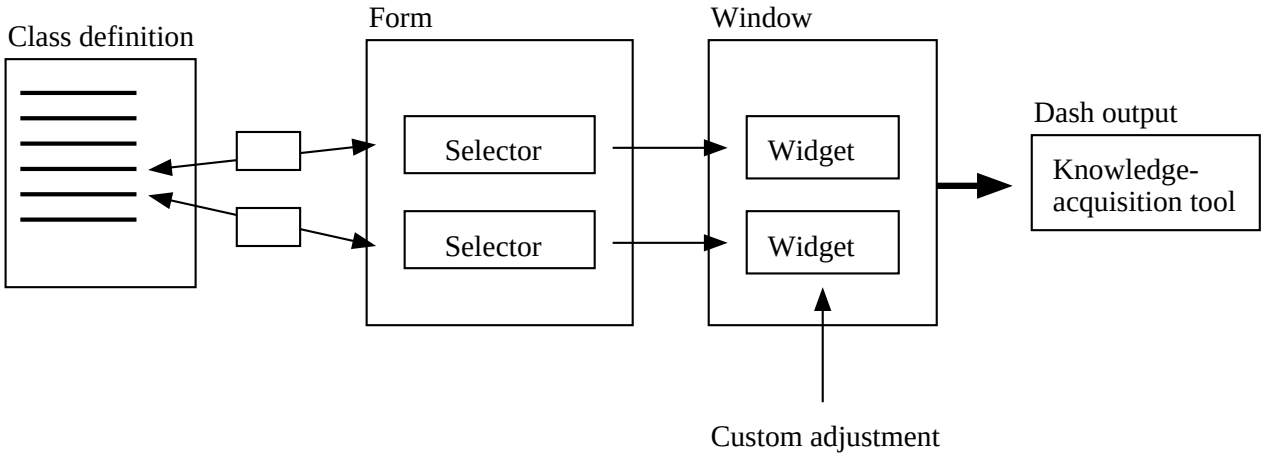
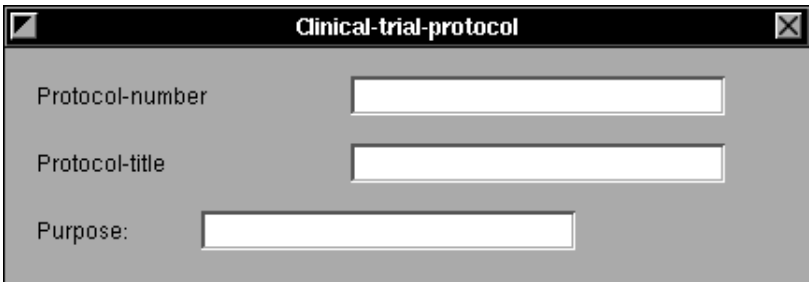
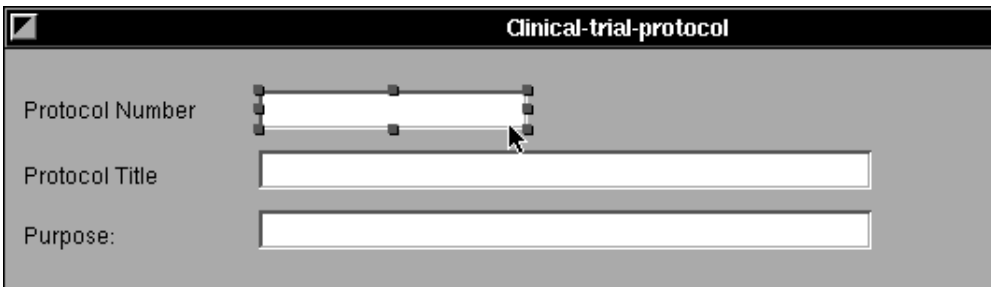


FIGURE 7. The mapping in DASH from class definitions to selectors and to widgets of the target tool. DASH maps slots to selectors (i.e., abstract user-interface widgets) to create an intermediate design representation. DASH then instantiates the selectors to widgets depending on the context in which the selectors are used, and on the current design policy.



(a)



(b)

FIGURE 8. Use of a graphical tool to custom tailor a form layout. (a) The default layout generated by DASH before adjustments. (b) The layout after manual adjustments. The developer can position and resize widgets, and can change labels.

4.3 Persistent Custom Adjustments

Normally, the developer designs incrementally the ontologies that DASH takes as input. It is extremely difficult to design a correct and complete domain ontology without defining representative instances of the ontology, and without testing the ontology with the application system. Some flaws in the domain ontology might be discovered relatively early in the development process. For instance, shortcomings uncovered in the use of the knowledge-acquisition tool may induce changes to the principal system design, and to the domain ontology (Figure 2). Examples of such changes include addition and deletions of slots in a class definition. Because domain ontologies may change after DASH has generated the corresponding knowledge-acquisition tool, DASH must support regeneration of the target tools. A developer may expend much effort custom tailoring a tool for an application domain, and it is important to preserve such manual changes so that they are not lost in the automatic regeneration of the knowledge-acquisition tool.

One approach to reduce this maintenance problem is to defer the custom tailoring of the tool until the domain ontology has stabilized. However, according to our experience, many shortcomings of the domain ontology are not apparent until the knowledge-acquisition tool—and even the application system—has been used for some time. The developer cannot defer custom adjustments of the tool until that stage. The DASH approach to this custom-tailoring problem is to store the changes persistently in a database, and to reapply the previous adjustments when DASH regenerates the knowledge-acquisition tool (see Figure 9). The advantage of this approach is that the developer can begin using DASH with a *temporary* ontology; it is not necessary to complete the ontology before generating the first version of the knowledge-acquisition tool.

As illustrated in Figure 9, DASH looks for previous custom adjustments, and, if they are present, reapplies them to the tool design. Next, DASH allows the developer to make additional custom adjustments to the tool. For instance, such adjustments might be required when the developer adds new classes to the ontology. Finally, DASH outputs a new version of the database for use in future regenerations of the target tool. Also, DASH provides an editor for the custom-tailoring database. The developer can use this editor to inspect, change, and withdraw entries in the database. An example of the use of this editor is when the developer previously has deleted a form field, and later wants to recall it. The developer can recall the field by removing the appropriate deletion record from the database.

4.4 Tool Design as Mappings

Dash can be viewed as a knowledge-based design system for knowledge-acquisition tools, because it incorporates design knowledge for such tools. Thus, DASH performs the task of an expert designer for domain-specific knowledge-acquisition tools. In many ways, the domain ontology that serves as input to DASH acts as a specification for the target tool. We can view the DASH architecture as a series of mappings that constitute the tool-design process. Dash maps the structure of the input ontology to the dialog structure using a set of design rules. Moreover, DASH maps the dialog structure together with the ontology to the selector representation, and then to the widget representation of the target tool (Figure 7). Finally, DASH maps the widget representation to a format that can be translated to C code, or interpreted by a tool run-time system (see Figure 3).

The initial version of DASH that we developed used a combination of procedural code

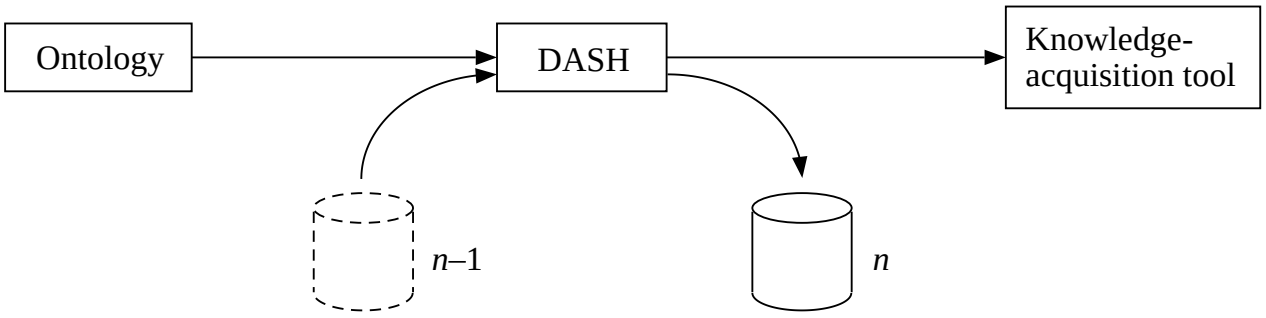


FIGURE 9. Use of the custom-tailoring database. DASH uses the database generated in the previous session ($n-1$) to install the previous adjustments. The developer can then make further adjustments. As part of the tool-generation process, DASH saves the current custom adjustments in a new version of the database (n).

and declarative mapping rules to design the target tools. After evaluating this version, we discovered a problem-solving pattern in many of subtasks of the tool-design process. Also, this version had grown increasingly difficult to maintain and extend. From the initial version of DASH, we abstracted the mappings, and implemented an explicit *mapping method* that can perform the mappings required in DASH (see Figure 10). We have configured this mapping method using object-oriented programming and production rules to implement the four most important mappings in the DASH architecture: from the ontology to the dialog structure, from the dialog structure to the selectors, from the selectors to the widgets, and from the widgets to the output tool definition.

Figure 11 shows an example of mappings used in DASH. The first instantiation of the mapping method maps slot definitions in the ontology, such as lists of instance pointers, to items in the dialog structure. The second method instantiation maps the dialog items to selectors. A slot that defines a list of instance pointers, for instance, would be mapped to a browser selector. In turn, the third method instantiation maps the browser selector to a set of widgets (i.e., a browser widget and add, delete, and edit button widgets). Finally, the fourth method instantiation maps the widgets to the output format.

The mapping task in DASH differs from other mapping tasks in PROTÉGÉ-II (Gennari, Tu, Rothenfluh & Musen, 1994) in that it requires structure mappings (i.e., mappings from a graph of objects to another graph of objects). In DASH, selectors, for instance, must be mapped to widgets as part of the tool-design process. However, there is not always a one-to-one correspondence between selectors and widgets. Moreover, selectors are sometimes linked to other selectors, and this linkage must be carried over to the widgets. (For instance, a button selector may be linked to a subform, and the corresponding button widget must be linked to the appropriate window representation.) The mapping method in DASH supports these mappings by automating some of the structure translations. This feature simplifies the mapping rules, and makes the configurations of the mapping method easier to maintain.

In summary, we have found the mapping method to be an extremely useful basic component in the DASH architecture. The principal advantage of the use of several instantiations of a mapping method in DASH is that the maintenance and extension tasks are simplified. It is possible to support additional languages for input ontologies by configuring a new in-

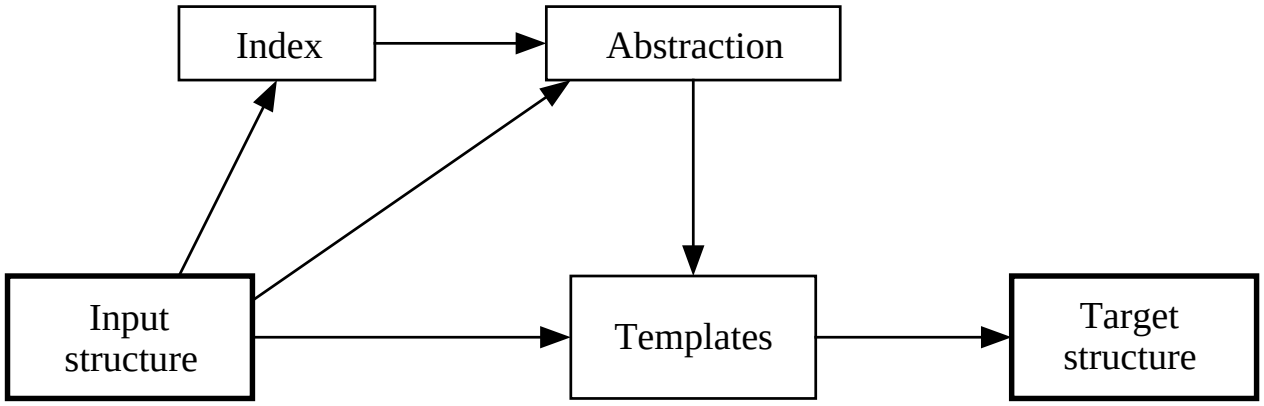


FIGURE 10. The structure of DASH's mapping method. The method indexes the input structure and abstracts mapping-specific features from the input structure using the index information. Next, the method uses a set of mapping rules to generate templates for the target structure. Finally, the method uses another set of rules to instantiate the target structure from the templates. Currently, we use four instantiations of this mapping method to define the DASH system.

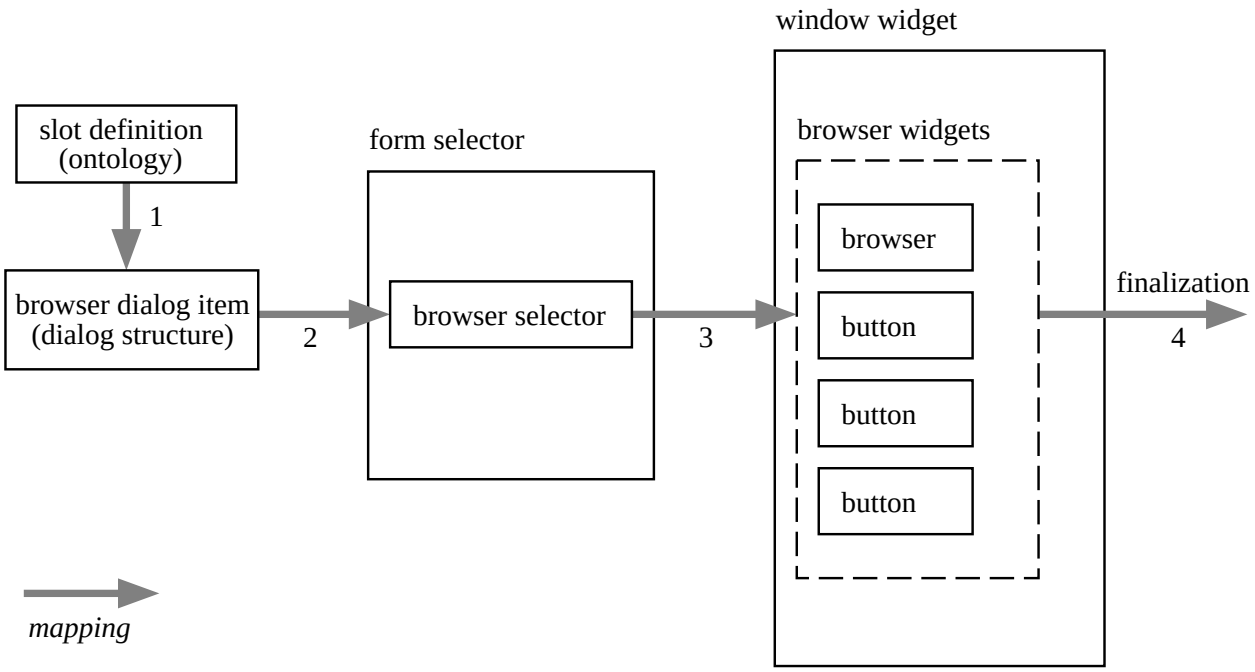


FIGURE 11. An example of a mapping chain in DASH. Each arrow represent an instantiation of DASH's mapping method.

```
(defclass company (is-a USER)
     (slot name (ka-specification browser-key) (type string))
     (slot telephone (type string))
     (slot fax (type string))
     (slot chairperson (type string))
     (slot employees (cardinality multiple)
                     (allowed-classes person) (type instance))
     (slot chapter11 (type boolean))
     (ka-specification top-level)
     (max-number-of-elements 100)
)

(defclass person (is-a USER)
     (slot name (ka-specification browser-key) (type string))
     (slot SSN (type integer))
     (slot address (type string))
     (slot city (type string))
     (slot state (type string))
     (slot zip-code (type integer))
     (slot telephone (type string))
)
```

FIGURE 12. Sample ontology that consists of class definitions for companies and persons. Each company has a list of employees, which are instances of the class person.

stantiation of the mapping method. Similarly, it is possible to add new definition languages for target knowledge-acquisition tools. The mapping method also allows for tool-supported maintenance and testing of DASH (Section 6).

5 Tool Generation: An Example

To illustrate the use of DASH, we shall discuss the generation of a sample knowledge-acquisition tool from a domain ontology. To simplify this example, we use a small ontology that is not particularly knowledge intensive. Let us assume that we have modeled a business domain, and have defined an ontology that consists of class definitions for *companies* and *persons*. We identify the relevant attributes for companies and persons, and we assume that each company has a list of employees. Figure 12 shows our class definitions for `company` and `person`. (Although we are using a textual ontology format for this example, the PROTÉGÉ-II environment includes a graphical editor for ontologies.) Note that the `ka-specification` class facet for `company` specifies that we need a top-level editor under the main menu for this class. Also note that the maximum number of `company` elements is 100. In both classes, the `ka-specification` facet for the slot `name` specifies that this slot should be the key in list browsers.

When we open this ontology with DASH, a dashboard window for the ontology will appear on the screen (see Figure 5). From the dashboard, we can inspect the tool design at various stages, and can issue tool-generation commands. Let us begin the tool-generation process by generating the dialog structure for the tool. We issue this command by clicking on the "Build Dialog Structure" button on the dashboard (see Figure 5). Figure 13 shows the resulting

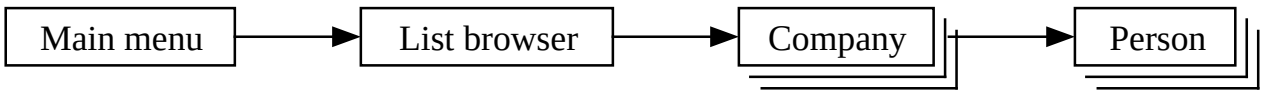


FIGURE 13. The dialog structure for the ontology shown in Figure 12. This dialog structure states that there is a list browser available under the main menu of the target tool. The list browser provides access to a list of companies, which the tool user can edit with a company editor. In turn, the company editor provides access to a list of persons, which the tool user can edit with a separate person editor.

dialog structure. In this example, the target tool will have one list browser under the main menu. This browser will provide access to a list of companies. Each company has a list of employees, which are instances of the class `person`.

The next step in the generation process is to generate the layout for the editors defined (by nodes) in the dialog structure. We generate the layout by clicking on the "Build Layout" button on the dashboard. After DASH has completed the layout design, we can custom tailor the user interface of the tool in NeXT's Interface Builder (see Figure 14). When we are satisfied with the layout, we save our modifications, and proceed with the tool generation (by clicking on "Build KA Tool"). DASH will now record our changes in the custom-tailoring database, and will produce the final definitions for the target tool.

At this point, we can compile the target tool to an executable system, or we can use a tool interpreter to run the tool. Figure 15 shows a sample session with the target tool. We have entered sample data in the tool, and have saved the result as instances of the `company` and `person` classes (see Figure 16). The instances `[kb1]` and `[kb2]` contain slot–value pairs with the information entered in the forms. Note that `[kb1]` has a list of employees with one item, `[kb2]`.

6 The DASH Development Environment

The DASH Development Environment (DDE) is a graphical environment that allows us to maintain and extend DASH. DDE supports the different configurations of the mapping method. Furthermore, DDE allows the DASH designer to run diagnostic tests on DASH, and to inspect graphically most of the intermediate representations in the tool-generation process.

6.1 BACKGROUND: DASH MAINTENANCE

Because we are using DASH to develop knowledge-acquisition tools on a regular basis in our laboratory, we must maintain and extend the system continuously. Although an explicit mapping method makes it easier to maintain DASH, DASH is still a complex system that can be difficult to support. Because of the complexity of the system and of its input data, it was initially often time consuming to isolate problems, and to make the appropriate changes to DASH. DDE has been designed to streamline the maintenance and further development of DASH by organizing the code for the configuration of the mapping method, and by supporting tests and inspection of test runs.

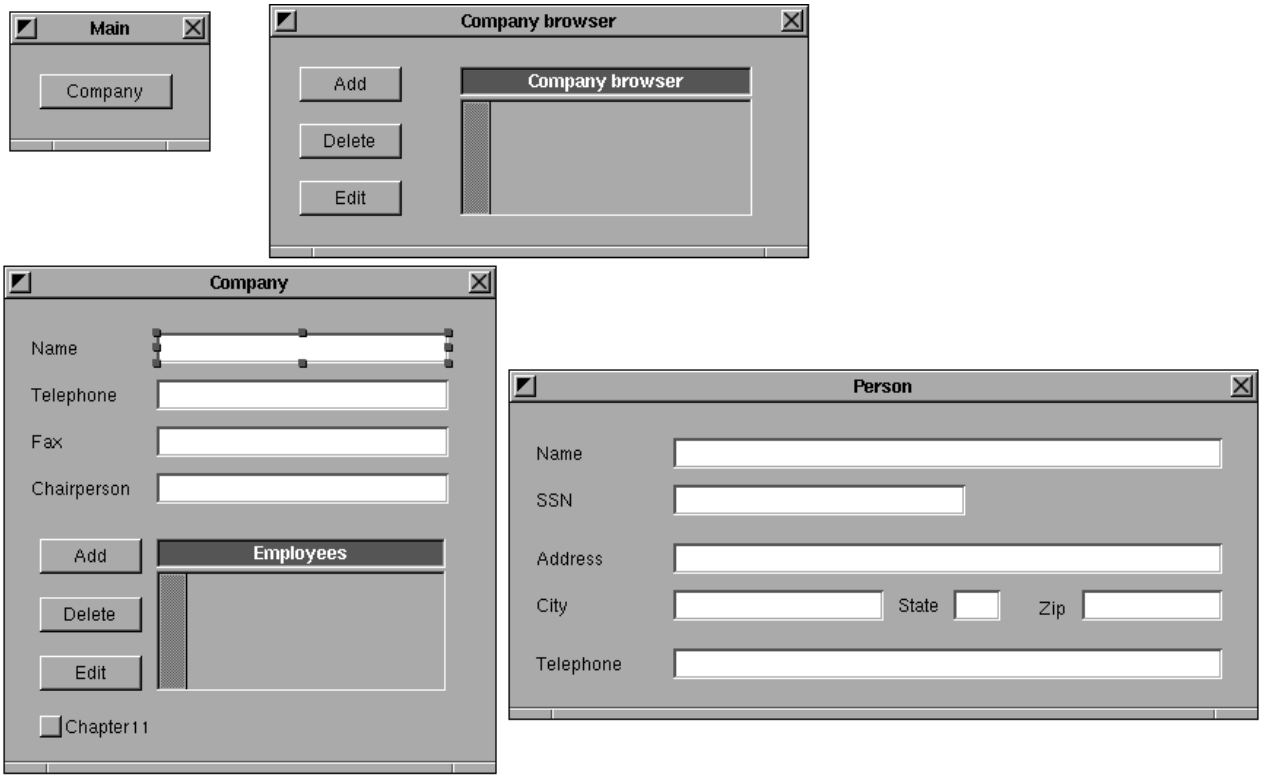


FIGURE 14. Custom tailoring of the layout for the sample knowledge-acquisition tool. The developer uses NeXT's Interface Builder to lay out the forms. Note that the main menu (upper left) has a button that provides access to the company browser (to the right of the main menu). The company form (lower left) contains a list browser for employees, which can be edited with the person form (lower right).

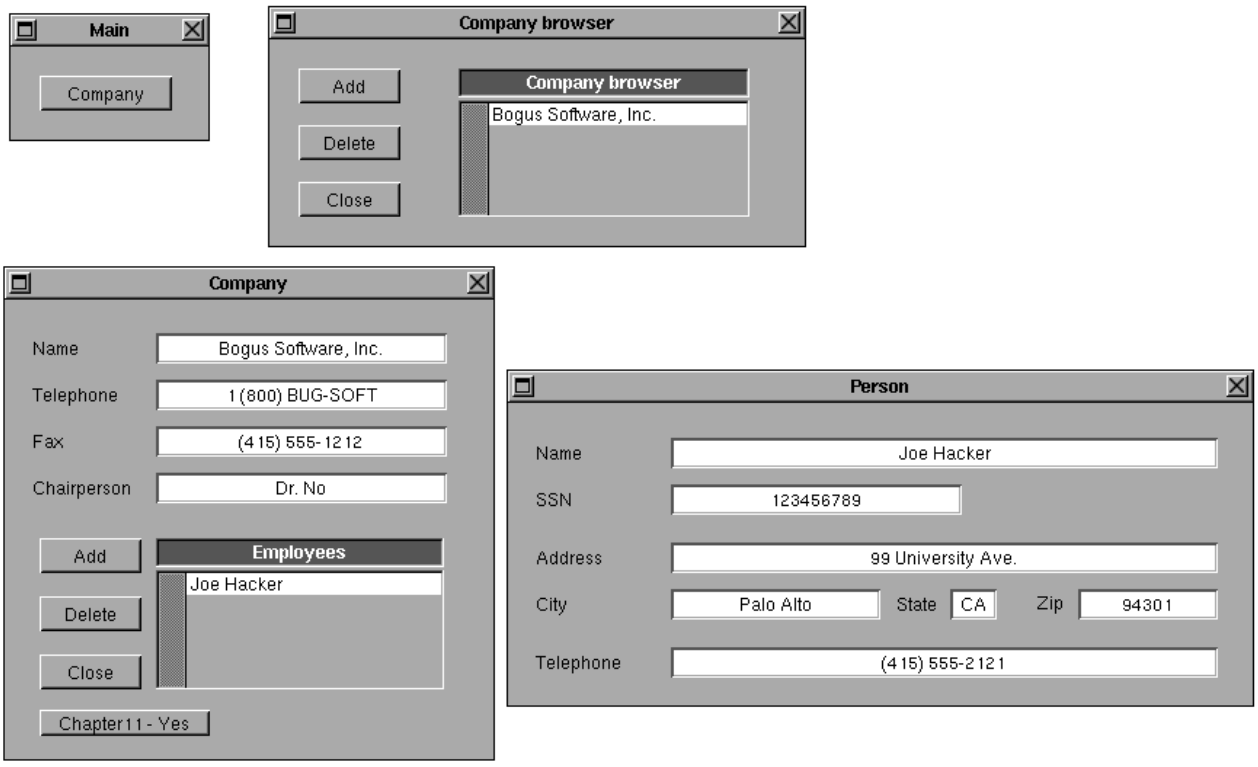


FIGURE 15. The generated knowledge-acquisition tool. We have entered sample data for one company with a single employee.

```
([kb1] of company
   (name "Bogus Software, Inc.")
   (telephone "1 (800) BUG-SOFT")
   (fax "(415) 555-1212")
   (chairperson "Dr. No")
   (employees [kb2])
   (chapter11 TRUE))

([kb2] of person
   (name "Joe Hacker")
   (SSN 123456789)
   (address "99 University Ave.")
   (city "Palo Alto")
   (state "CA")
   (zip-code 94301)
   (telephone "(415) 555-2121"))
```

FIGURE 16. The output instances from the knowledge-acquisition tool in Figure 15. The instances `[kb1]` and `[kb2]` are of the classes `company` and `person`, respectively (Figure 12), and contain the information entered in the forms for company and person, respectively.

6.2 The Developer's Interface

When the DASH developer starts DDE, the main menu will appear on the screen (Figure 17). This menu illustrates the DASH architecture, and provides access to the functions of DDE. The upper part of the window shows a data-flow model of the design process in DASH. The user can click on the nodes in the data-flow diagram (such as input ontology, dialog designer, and so on) to edit the instantiations of the mapping method (Section 4.4) in a configuration tool (Figure 18). The developer instantiates the mapping method by providing the appropriate subclasses, methods, and rules. (The developer must design a method configuration that accomplishes the tool-design task through these definitions.)

The buttons in the lower left-hand corner of the window provide access to the selector and widget ontologies, respectively. The button group in the lower center of the window provides access to various subsystems of DASH that do not fit neatly into the dataflow model, such as user preferences, window layout algorithms, and functions related to the custom-tailoring database. The buttons in the lower right-hand corner control the source files of DASH. Finally, the button labeled "Test" in the upper right-hand corner provides access to a test module for DASH (Section 6.3). Note that DDE is application specific, because its user interface incorporates the DASH architecture and makes assumptions about DASH's mapping method.

Figure 18 shows the graphical configuration tool for the mapping method. This tool allows the DASH developer to configure instantiations of the mapping method. The lower half of the mapping panel shows the structure of the mapping method (see Figure 10). The DDE user can access definitions (e.g., classes, methods, and rules) of the components of the mapping method by clicking on the nodes in the data-flow graph. Also, the developer can place definitions that do not fit neatly into the method's data-flow model (e.g., utility functions) under the button for miscellaneous constructs.

By clicking on one of the nodes in the data-flow model on the mapping configuration panel, the DDE user can add new constructs (through a browser). These constructs bottom out in Common Lisp extended with CLOS and production rules (Steele Jr., 1990; Keene, 1989). Figure 19 shows a sample rule used in the method configuration for the widget composer. This rule states that selector components of the type `form-entry-field` where the `field-type` is `text` should be mapped to a widget set, which consists of a display widget that shows the field label (which is constructed by another rule) and a `text-field` widget. Also, the rule, if activated, constructs some additional information that is needed by the `text-field` widget, such as the origin of the widget and references to the ontology (i.e., the class and slot that the text field corresponds to).

In addition to configuration of the mapping method, the DDE main menu provides access to class definitions for selectors and widgets. The selector- and widget-ontology buttons open browsers for these ontologies. Figure 20 shows a graphical tool that allow the DDE user to edit the widget ontology. (DDE provides a similar tool for the selector ontology.) Each node in the graph represent a widget class (which is implemented as a CLOS class), and the links among the nodes represent *is-a* links. For instance, a `browser-add-button` is a subclass of a `browser-button,` which in turn is a subclass of a `push-button`. By clicking on the nodes in the graph, the user can edit the class definitions.

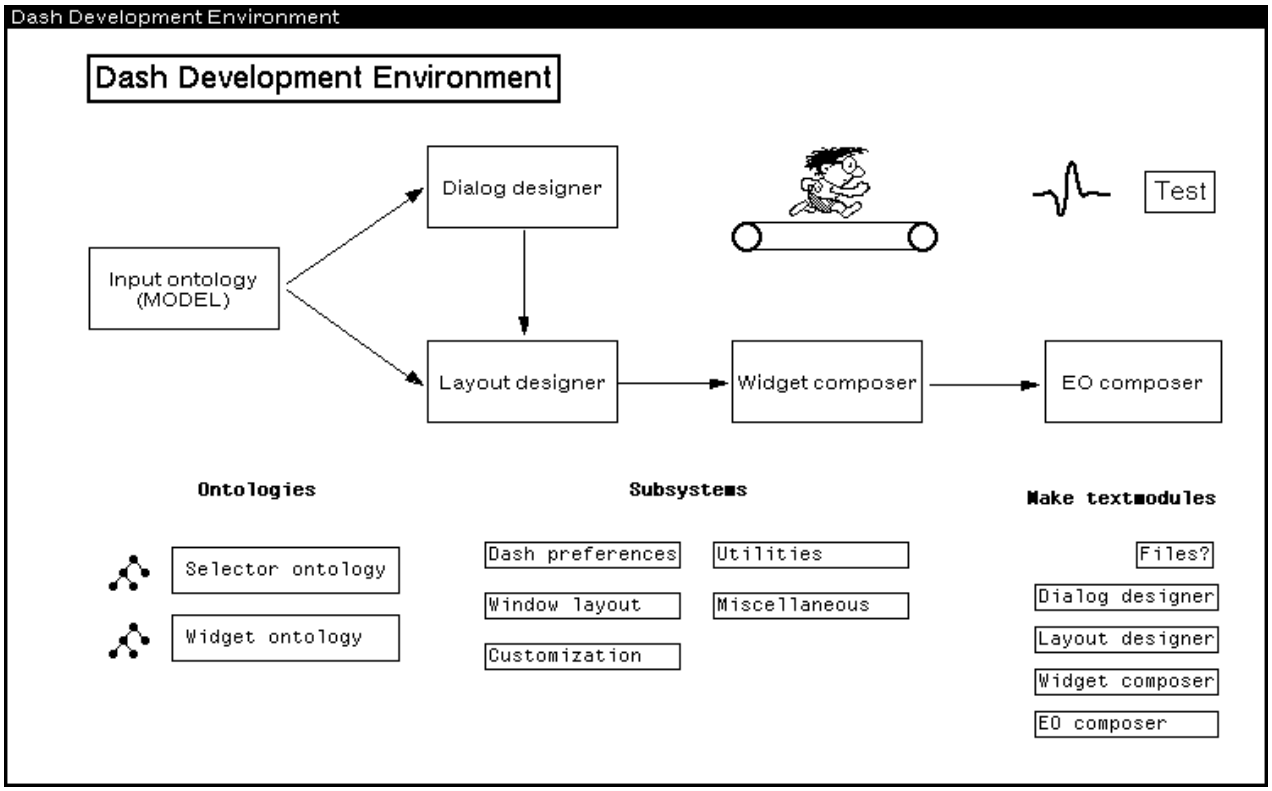


FIGURE 17. The main menu for DDE. This window illustrates the data-flow relationships in the DASH architecture. The data-flow diagram shows that the domain ontology is the input to the dialog designer, and that the layout designer uses the dialog structure produced by the dialog designer together with the domain ontology to produce selectors. The widget composer uses these selectors to design the widgets, which are finalized to the output structure (by the EO composer). The DDE main menu provides access to method configurations for the mapping method. Also, the user can access ontologies and subsystems of DASH, can manage files, and can run diagnostic tests of DASH.

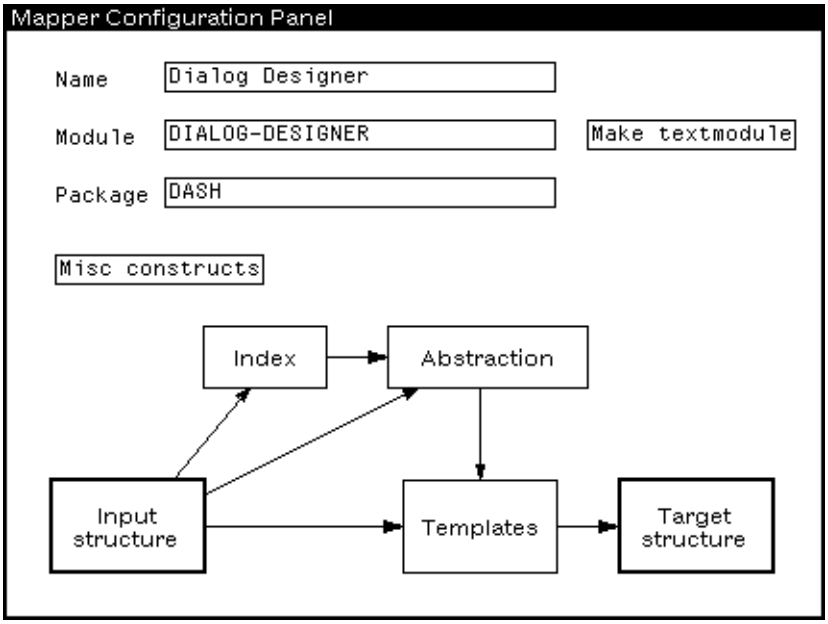


FIGURE 18. Configuration of the mapping method in DDE for the dialog designer. The DASH developer can click on the boxes in the data-flow diagram and provide the definitions required to configure the mapping method (e.g., subclasses, methods, and rules).

SEdit WC-WIDGET-FORM-4A Package: DASH

```
(defrule wc-widget-form-4a
        ((templ :whichis wc-templ-item)
         (component :in-eval (mapper::source-items templ)))
    (and (typep component 'form-entry-field)
         (eq (query component field-type) :text))
  (conclude (path templ widgets)
            ▮(,@(query templ display-widgets)
              ,(make-instance 'text-field
                 :label-string
                 ""
                 :model-reference
                 (query component model-reference)
                 :origin
                 (origin component)))))
```

FIGURE 19. Sample rule definition from the configuration of the mapping method for the widget composer. This rule maps `form-entry-field` selectors for text to `text-field` widgets with labels.

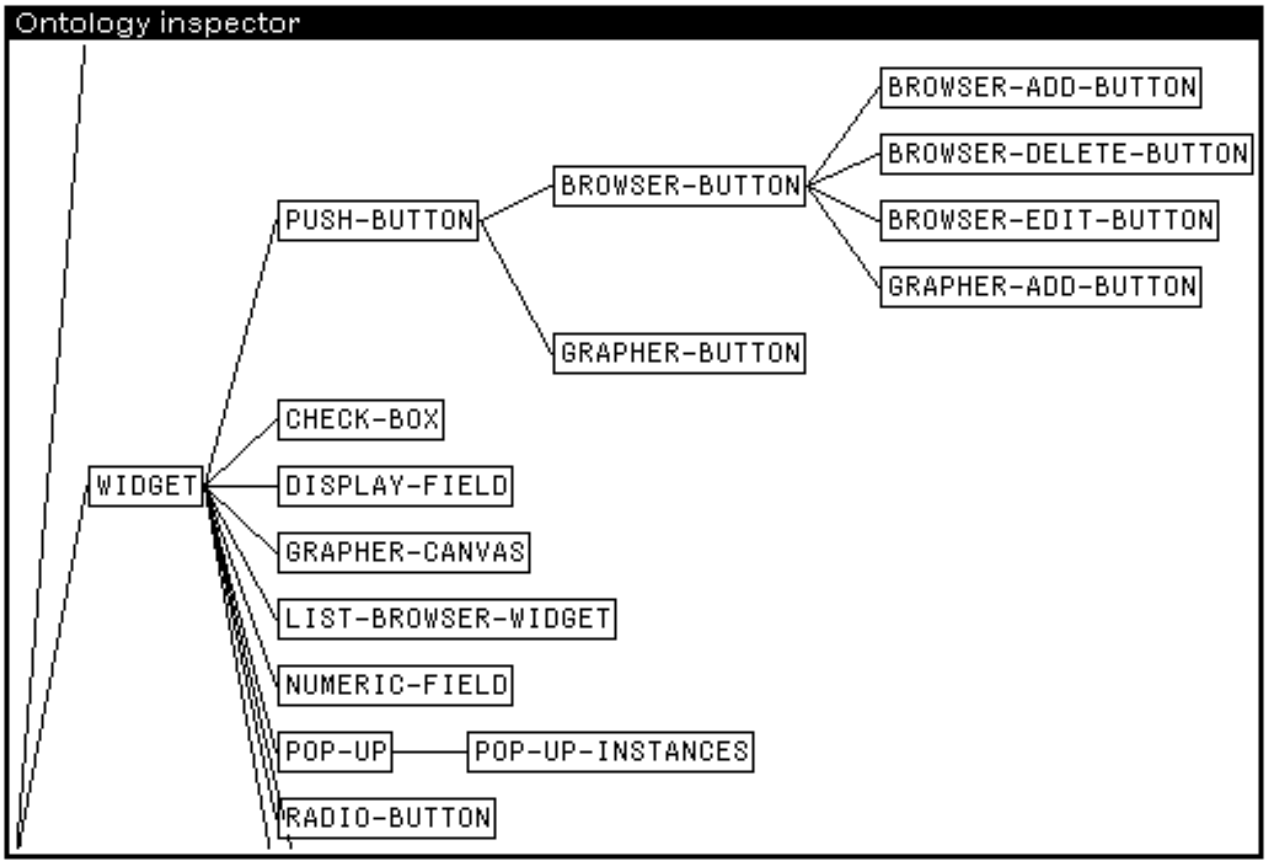


FIGURE 20. DDE allows the DASH developer to inspect and edit the widget ontology.

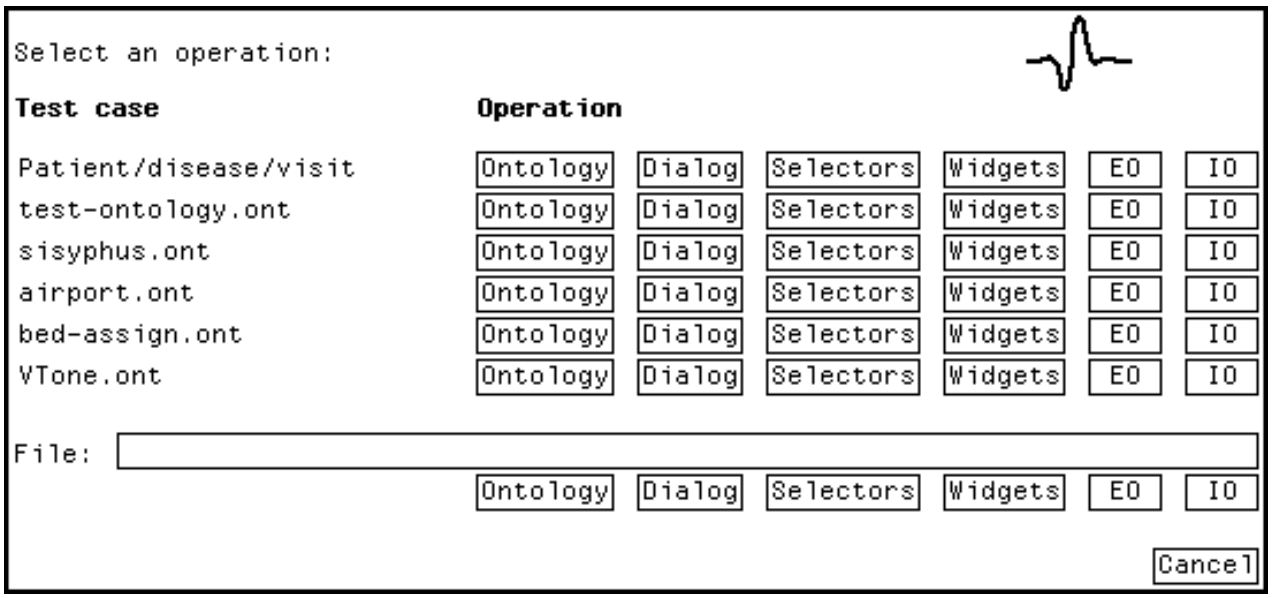


FIGURE 21. The test panel in DDE. The DASH developer uses this panel to run diagnostic tests on DASH. By clicking on one of the operation buttons, the user can run the test case up to the point selected.

6.3 SUPPORT FOR TEST AND INSPECTION

DDE provides functions for running test cases on DASH, and for monitoring the results. DDE runs DASH in a controlled test mode, and allows the DASH developer can inspect and verify intermediate results. Currently, DDE does not support automatic verification of the intermediate results (i.e., consistency checking). Instead, DDE is designed to provide the developer powerful inspectors for the result. This looking-glass approach allows the developer to better understand where and why problems occurred.

The first step in maintenance and debugging is often to reproduce a problem reported under controlled circumstances. Therefore, DDE provides a graphical interface for running test cases. The button labeled "Test" on the DDE main menu brings up the test panel (Figure 21). This panel allows the user to run test cases based on typical input ontologies. By clicking on the appropriate operation button, the user can run DASH on the corresponding test case up to the point selected. For instance, by clicking on the "Selectors" button for the `sisyphus.ont` test case, the user will instruct DASH to generate selectors from the Sisyphus domain ontology (Linster, 1992). By default, the test panel shows a set of common test cases. The DDE developer can add new cases to the test panel readily by specifying the ontology files to use. Also, the DDE user can specify explicitly any ontology file as a test case. This functionality is useful if a DASH user encounters problems with a specific domain ontology. The DASH developer can then isolate the problem by running this ontology as a test case, and inspecting intermediate results in DASH.

When the test operation is completed, DDE will open a window that provides an interactive test protocol. The DDE user can then interact with this protocol to inspect the intermediate data structures produced by DASH. Figure 22 shows a sample test protocol. By clicking on the "Input ontology" button, the user can inspect the input to the dialog designer. Figure 23

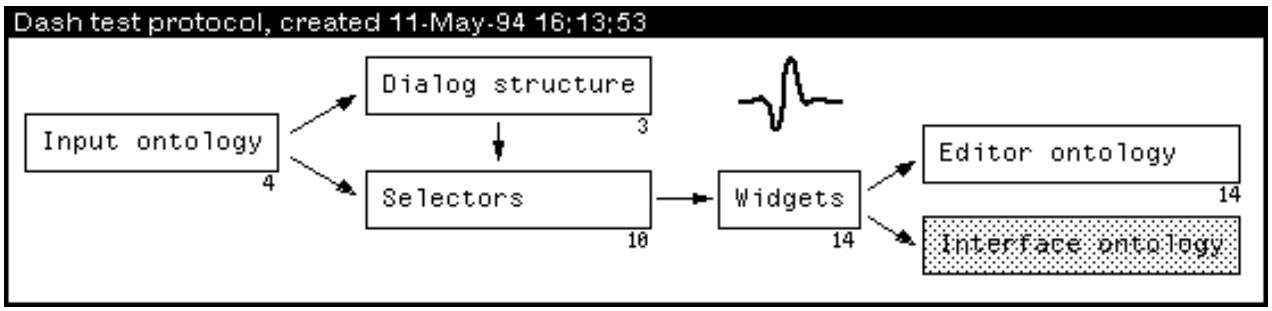


FIGURE 22. An interactive test protocol from a test case. The DASH developer can inspect the state of DASH after a test run by clicking on boxes and arrows in the protocol window. The numbers below the boxes represent the number of items generated by DASH for the test case (e.g., 10 selectors and 14 widgets have been generated here).

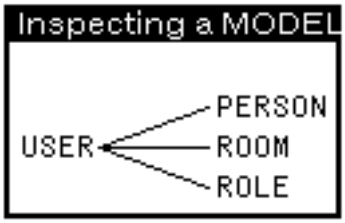


FIGURE 23. The input ontology for the test case. `USER` is the top-level class.

shows the ontology for the Sisyphus room-assignment problem. This ontology defines the classes `PERSON`, `ROOM`, and `ROLE`.[1] Moreover, the user can inspect the internal representation of the clsses by clicking on the appropriate items in the graph.

By clicking on the "Dialog structure" button, the user can inspect the dialog structure generated from the input ontology. Figure 24 shows the dialog-structure inspector for the test case. In this case, the dialog structure consists of the main menu, a list browser, and a form for *roles* as defined in the ontology. (The classes `PERSON` and `ROOM` in the ontology do not result in forms in the dialog structure, because the `ka-specification` facet for these classes are specified as `ignore`.) The user can inspect the representation for the items in the dialog structure by clicking on them in the graph. Thus, the DDE user can use the dialog-structure graph and the inspectors to verify that the mapping from the input ontology to the dialog structure is working correctly.

The DDE user can examine the selector and widget structures generated by DASH by clicking on the "Selectors" and "Widgets" buttons on the test protocol, respectively. Figure 25 shows the selector structure for this test case. The main form has one `FORM-BUTTON` selector, and the role-browser window has a single `FORM-LIST-BROWSER` selector. The role form contains three `FORM-SELECTION` selectors (which allows the tool user to input Boolean values), and one `FORM-ENTRY-FIELD` selector (which allows the user to input the role name).

[1] In this domain ontology, the term *role* describes the professional role of an office worker (e.g., secretary, and manager).

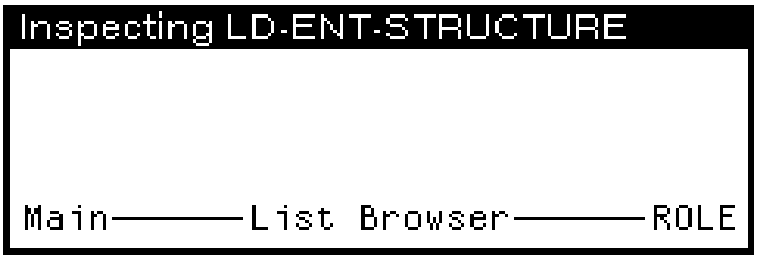


FIGURE 24. The dialog structure for the test case.

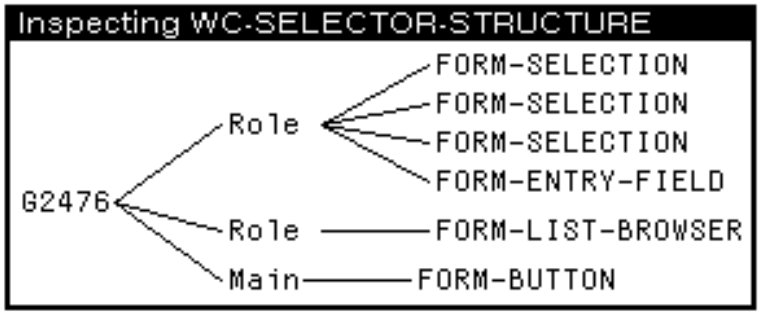


FIGURE 25. The selectors for the test case.

Figure 26 shows the widget structure after DASH has mapped the selectors to widgets. As shown in this inspector, DASH has mapped the `FORM-SELECTION` selectors to `CHECK-BOX` widgets, and the `FORM-ENTRY-FIELD` selector to a `TEXT-FIELD` widget *and* a `DISPLAY-FIELD` widget, for example. (The rule shown in Figure 19 is responsible for the latter mapping.) Note that DASH has expanded the `FORM-LIST-BROWSER` selector to a set of four widgets, which constitutes the browser.

By clicking on the nodes in the selector and widget structure windows, the DDE user can bring up inspectors for the underlying data structures. Figure 27 shows a sample inspector for a check-box widget. The left column consists of slot names in the check-box widget class, and the right column consists of slot values for the widget instance selected. The underlying lisp system provides the implementation of this `clos` inspector. Therefore, the user can inspect recursively structures shown in the inspector window.

The final generation step is to translate the window structure to the output format. Currently, we are using the *editor-ontology* (EO) format as the output of DASH. Figure 28 shows the inspector for the resulting EO structure. (Note that this format uses generic names for windows and widgets.) Again, the DDE user can click on nodes and inspect the content of the underlying representation. The output of DASH is a text file that describes instances of the editor ontology; DASH uses a straightforward printing routine to produce the appropriate syntax for the EO structure. Furthermore, DDE allows the user to inspect the output of DASH in the textual format.

In summary, the interactive test protocol and its inspectors is a powerful tool for the debugging of DASH. A typical debugging strategy includes (1) the selection of an appropriate

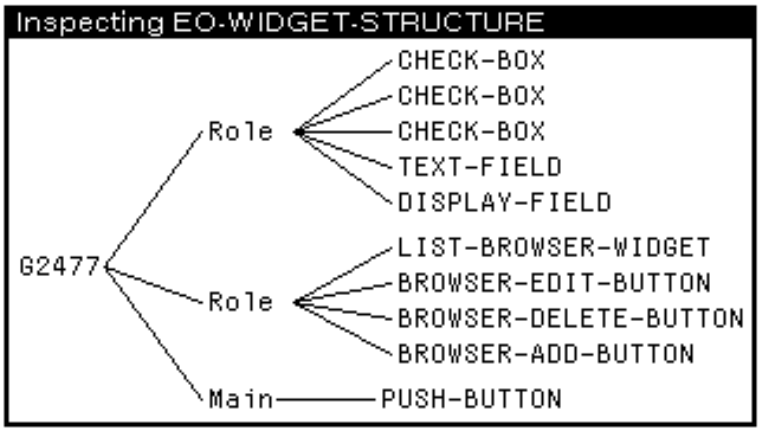


FIGURE 26. The widgets for the test case.

Inspecting a check-box

dash::origin	#<dash::origin-slot-link 346,145540>
dash::extra-id	"** slot not bound **"
dash::deleted	nil
dash::added-component	nil
dash::x-coord	20
dash::y-coord	80
dash::width	100
dash::height	15
dash::parent-widget-group	"** slot not bound **"
dash::label-string	"Large room"
dash::label-font	"** slot not bound **"
dash::label-pixmap	"** slot not bound **"
dash::label-type	:string
dash::model-reference	#<dash::model-reference 346,145514>

FIGURE 27. Inspection of a check-box widget. For instance, the slots `x-coord`, `y-coord`, `width`, and `height` represent the widget geometry. The slot `label-string` represent the label for the check box (in this case, the slot value is "Large room").

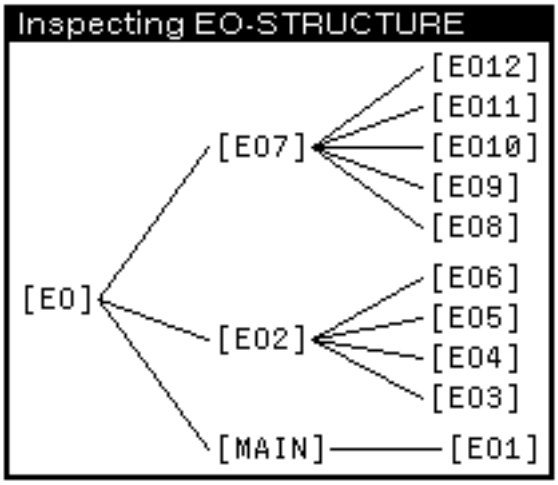


FIGURE 28. The resulting EO structure for the test case. This graph provides an overview of the EO representation. The node labeled `[EO]` represents the top-level EO structure. This structure contains a list of the target-tool windows, in this case, `[EO7]`, `[EO2]`, and `[MAIN]`. Furthermore, each of these windows contains a list of its widgets. (For instance, the window `[EO7]` contains the widgets `EO8` through `[EO12]`.)

test case (ontology), (2) the test execution of DASH to an appropriate point, and (3) the inspection of the intermediate data structures produced by DASH. The DDE user can then use a divide-and-conquer strategy to locate the problem by inspecting the results before and after the appropriate mapping steps. Moreover, the DDE user can click on the arrows in the test-protocol panel (Figure 22) to inspect the internal steps in each instantiation of the mapping method, and to further narrow down the problem.

6.4 DDE SUMMARY

We are currently using DDE on a regular basis, and we have found DDE to be extremely useful for the maintenance of DASH. Not only have the number of bugs decreased, but the time required to correct a bug has been shortened significantly. The most important factors for this improvement are (1) the increased level of organization that the mapping method and its graphical configuration tool provide, and (2) the enhanced visibility and transparency of the internal DASH data structures the inspectors provide. Because DDE incorporates a model of the DASH architecture in its user interface, DDE allows the DASH developer to navigate through a complex application readily, and to understand the behavior of the system by inspecting the results of the execution based on the architectural model. Although we are currently using DDE for general DASH development, it is possible to use DDE to extend DASH to support the functionality required for a specific domain. For instance, new (domain-specific) design rules can be added to DASH to extend the knowledge-acquisition support for a domain.

The DDE approach to providing application-specific development and maintenance environments is applicable potentially to other types of software. Analogous to domain-specific knowledge-acquisition tools, application-specific development environments require a significant tool-development effort for a single domain. However, metatools similar to DASH can be

used generate such application-specific environments given an architectural description of the target system, for instance.

7 Discussion

The role of DASH within the PROTÉGÉ-II framework is to assist the developer in the design of domain-specific tools that domain specialists can use to edit instances of ontologies (i.e., to perform the detailed design of the system rather than the principal design; see Figure 2). Other parts of the PROTÉGÉ-II architecture support the definition of ontologies, and the reuse of problem-solving methods (Puerta *et al.*, 1992). Our motivation for using an ontology-driven approach to the generation of knowledge-acquisition tools in DASH is that we want to provide the PROTÉGÉ-II users with a general, yet easy-to-use metatool for generating domain-specific knowledge-acquisition tools.

Compared to method-driven generation of knowledge-acquisition tools, such as the Spark approach (Marques *et al.*, 1992), ontology-driven generation allows the metatool to generate a domain-specific, rather than method-specific, target tool. Domain-specific target tools incorporate domain concepts familiar to domain specialists (who are typically nonprogrammers), rather than generic terms defined by programmers. Our approach differs from approaches to model-based user-interface design in that DASH is designed to produce a complete application (the target tool) rather than produce a user interface for an application program developed independently. Moreover, the DASH architecture provides features for the high-level design of the dialog structure, and for the persistent storage of custom adjustments.

Initially, we designed DASH to generate domain-specific knowledge-acquisition tools from domain ontologies (rather than from other types of ontologies). Nevertheless, we have used DASH to generate tools for other tasks. For instance, we have experimented with the generation of method-specific tools from ontologies of problem-solving methods. Such tools are intended for developers (who are programmers) rather than for domain specialists, because the tools incorporate extremely generic abstraction rather than application-specific terms (Musen, 1989*a*).

The models that developers define as domain ontologies for application systems are not always unambiguous for the purpose of generating knowledge-acquisition tools. Although domain ontologies provide most of the information required to generate basic knowledge-acquisition tools, developers must annotate the ontologies with information related to knowledge acquisition. For instance, the developer must indicate the classes of domain concepts that the target-tool user requires editors for directly under the main menu of the tool. In the current version of DASH, the developer provides this information as slot and class facets in the ontology. The disadvantage of this approach is that the developer adds knowledge-acquisition information to the domain ontology, which is supposed to provide a clean model of the domain, and thereby makes the ontology impure. Thus, DASH takes as input a combined model of the domain and of the presentation and editing of instances of domain concepts. Annotating the ontology with knowledge-acquisition information is a relatively simple technique for providing this specification. An alternative approach, which we are considering currently, is to specify this information separately from the ontology, and to use this information together with the ontology as input to DASH. The advantage of the latter approach is that the developer can generate several alternative tools from the same ontology.

The generation of a high-level dialog structure as an intermediate tool-generation step is

a salient feature of DASH. To generate a nontrivial tool, it is essential to establish the relationships among the editors and windows of the target tool before instantiating their layouts, because the layout-design process requires this information to generate appropriate buttons and browsers. The dialog structure not only is important for the subsequent generation steps, but also is useful for the developer. By examining the dialog structure graphically, the developer can understand the structure of the target tool. In our experiments with DASH, the developer has often discovered flaws in the domain ontology by inspecting the dialog structure.

Persistent storage of custom adjustments is an important feature that enables developers to refine the domain ontology incrementally. Without such persistent adjustments, the developer cannot generate a new tool from a modified ontology readily. Although DASH's approach of using a custom-tailoring database works well in most cases, there are situations where discrepancies between the saved changes and the current domain ontology might occur. Because DASH matches the previous entries in the custom-tailoring database against the new widgets generated by the layout designer, some records in the database might be obsolete due to changes in the domain ontology. For instance, this situation arises when the developer removes classes and slots from the ontology. In this case, DASH ignores the corresponding information in the database. The other principal difficulty arises when the developer adds new classes and slots to the ontology. Here, DASH uses the information available in the database as much as possible, and applies a default layout algorithm for the new widgets.

An important question for metalevel tools is how general the system is in terms of the type of target tools the system can generate. To test the generality of the DASH approach for generation of knowledge-acquisition tools, we defined an ontology that models the domain of ontologies. In other words, we defined metaclasses for the ontology. For example, we defined the class `ontology-class` as the class of classes in the ontology. Each instance of `ontology-class` includes a list of slot instances (of the class `ontology-slot`), as well as other class attributes. We then applied DASH to this metalevel ontology. With this input, the output of DASH is a knowledge-acquisition tool for the domain of ontologies—that is, the output is an ontology editor. Next, we used this ontology editor to define a new ontology.

To challenge the system, we entered the metalevel ontology in the ontology editor. The output of the ontology editor is a set of instances of `ontology-class, ontology-slot,` and so on. Because these ontology instances are expressed in the instance syntax, rather than in the class-definition syntax, we used a small program (about one page of code) to translate the instances to the appropriate syntax. After running this program on the instances produced by the ontology editor, we got the original metalevel ontology. To complete the circle, we again used this ontology as input to DASH.

We have used DASH to generate several knowledge-acquisition tools for different domains and tasks. The domains and tasks that we have modeled at this point include assignment of office workers to rooms, assignment of patients to hospital beds, configuration of elevators, and management of clinical-trial plans for AIDS treatment. The room-assignment and elevator-configuration problems are defined in the Sisyphus suite of standard problems for knowledge acquisition and problem solving (Linster, 1992; Marcus, Stout & McDermott, 1988). The knowledge-acquisition tool for the clinical-trial domain is designed for a significant application system (Musen, Carlson, Fagan, Deresinski & Shortliffe, 1992). Table 1 illustrates the size of these knowledge-acquisition tools. DASH has provided significant support for the development of these tools, and has made it feasible to use domain-specific tools in these projects.

TABLE 1. Sizes of knowledge-acquisition tools generated by DASH. The figures refer to the number of items for each tool and category.

Domain	Ontology classes	Dialog-structure items	Selectors	Widgets	Windows
Sisyphus 1	3	3	10	14	3
Gate allocation	6	7	26	43	7
Bed assignment	5	5	19	29	5
UHaul	12	15	65	114	15
Sisyphus 2 (VT)	78	275	—	—	38
T-HELPER	76	133	—	—	25

8 Conclusions

The generation of knowledge-acquisition tools from domain ontologies is an approach where the developer *reuses* domain ontologies developed originally for application systems in the generation of target tools. Tool generation from domain ontologies is an appropriate approach, because developers can apply it in practical system development. To generate nontrivial tools, we must extend the basic mapping from data types in the class definitions to user-interface components with other functions, such as dialog design and custom adjustments. Persistent custom adjustments provide another important function that makes the metatool useful in iterative system development where the ontology is changing during the development process. DASH demonstrates how we can combine these functions to a practical metatool.

From the developers' perspective, DASH is easy to use, because it takes advantage of the same ontology that is used in the run-time system, rather than requiring a separate, parallel specification for the target tool as input. As a metatool, DASH combines generality in terms of target tools with ease of use for the developer. The design support provided by DASH's dialog-designer module helps developers with the important task of designing the overall structure of the knowledge-acquisition tool. The layout-designer module automates much of the tedious task of implementing the user interface at the selector and widget levels. Finally, DASH allows the developer to custom tailor the target tool.

Although DASH supports forms and list browsers in the tools it generates, a drawback of the current version of DASH is that there is no support for graph editors, and for corresponding palettes. We are working on an extended version of DASH that will overcome these deficiencies by incorporating appropriate mappings from the input ontology to instantiations of a generic graph editor in the target tool. Despite these shortcomings, we believe that the generation of knowledge-acquisition tools from domain ontologies is a flexible and viable approach, especially because we have used successfully the DASH architecture to generate several knowledge-acquisition tools for different domains, and to custom tailor these tools to their applications.

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 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 Egar , John Gennari , Thomas Rothenfluh , and Samson Tu for valuable discussions on DASH. We are grateful to Lyn Dupré for providing editorial assistance.

On-line information about PROTÉGÉ-II and DASH is available through a World-Wide-Web (WWW) service (`http://camis.stanford.edu/protege/`).

References

de Baar, D. J. M. J., Foley, J. D. & Mullet, K. E. (1992). Coupling application design and user interface design. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '92)*, pp. 259–266, Monterey, CA. ACM, New York.

Eriksson, H. (1992). Metatool support for custom-tailored domain-oriented knowledge acquisition. *Knowledge Acquisition*, **4**, 445–476.

Eriksson, H. (1993). Specification and generation of custom-tailored knowledge-acquisition tools. In *Proceedings of the Thirteen International Joint Conference on Artificial Intelligence, IJCAI'93*, pp. 510–515, Chambéry, Savoie, France.

Eriksson, H. & Musen, M. A. (1993). Metatools for knowledge acquisition. *IEEE Software*, **10**, 23–29.

Gennari, J. H., Tu, S. W., Rothenfluh, T. E. & Musen, M. A. (1994). Mapping domains to methods in support of reuse. In *Proceedings of the Eighth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pp. 24.1–24.20, Banff, Canada.

Johnson, J. (1992). Selectors: Going beyond user-interface widgets. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '92)*, pp. 273–279, Monterey, CA. ACM, New York.

Karbach, W., Linster, M. & Voss, A. (1990). Models, methods, roles and tasks: Many labels—one idea? *Knowledge Acquisition*, **2**, 279–299.

Keene, S. E. (1989). *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS*. Massachusetts: Addison-Wesley.

Linster, M., Ed. (1992). *Sisyphus'92: Models of Problem Solving*, Technical Report 630, Gesellschaft für Mathematik und Datenverarbeitung (GMD), St. Augustin, Germany.

Marcus, S., Stout, J. & McDermott, J. (1988). VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, **9**, 95–112.

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

McDermott, J. (1988). Preliminary steps toward a taxonomy of problem-solving methods. In Marcus, S., Ed., *Automating Knowledge Acquisition for Expert Systems*, chapter 8, pp. 225–256. Boston, MA: Kluwer Academic Publishers.

Musen, M. A. (1989*a*). Conceptual models of interactive knowledge acquisition tools. *Knowledge Acquisition*, **1**, 73–88.

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

Musen, M. A., Carlson, R. W., Fagan, L. M., Deresinski, S. C. & Shortliffe, E. H. (1992). T-HELPER: Automated support for community-based clinical research. In *Proceedings of the Sixteenth Annual Symposium on Computer Applications in Medical Care*, pp. 719–723, Washington, D.C.

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

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

NeXT (1990). *NeXTstep Concepts*, NeXT Computer, Redwood City, CA.

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

Puerta, A. R., Tu, S. W. & Musen, M. A. (1993). Modeling tasks with mechanisms. *International Journal of Intelligent Systems*, **8**, 129–152.

Steele Jr., G. L. (1990). *Common LISP: The Language*, second edition. Bedford, MA: Digital Press.

Steels, L. (1990). Components of expertise. *AI Magazine*, **11**, 28–49.

Tu, S. W., Kahn, M. G., Musen, M. A., Ferguson, J. C., Shortliffe, E. H. & Fagan, L. M. (1989). Episodic skeletal-plan refinement based on temporal data. *Communications of the ACM*, **32**, 1439–1455.