The Knowledge Engineering Review, Vol. 8:1, 1993, 27–47

Conceptual models for automatic generation of knowledge-acquisition tools

HENRIK ERIKSSON* AND MARK A. MUSEN

Medical Computer Science Group, Knowledge Systems Laboratory, Stanford University School of Medicine, Stanford, CA 94305-5479, USA

Abstract

Interactive knowledge-acquisition (KA) programs allow users to enter relevant domain knowledge according to a model predefined by the tool developers. KA tools are designed to provide *conceptual models* of the knowledge to their users. Many different classes of models are possible, resulting in different categories of tools. Whenever it is possible to describe KA tools according to explicit conceptual models, it is also possible to edit the models and to instantiate new KA tools automatically for specialized purposes. Several *meta-tools* that address this task have been implemented. Meta-tools provide developers of domain-specific KA tools with generic design models, or *meta-views*, of the emerging KA tools. The same KA tool can be specified according to several alternative meta-views.

1 Introduction

Numerous knowledge-acquisition (KA) tools have been implemented in research laboratories. From a research point of view, implementations of tools provide the means to test KA models and methods in realistic situations. In many ways, the creating of these KA tools is still a research issue, and sometimes is an art. Nevertheless, it is desirable to classify and understand more clearly the principles behind various KA tools so that we can, for instance, outline new generations of tools.

One way of classifying KA tools is to group them according to the *conceptual model* that they present to their users (Musen, 1989b). A conceptual model, in this context, is the metaphor for the user interaction; for instance, it is the way in which knowledge is entered, edited and presented. (We distinguish such conceptual models for KA tools from *conceptual domain models*, which describe the experts' view of the domain and the relevant domain knowledge.) Examples of conceptual models include *symbol-level*, *method-based* and *task-based* conceptual models. KA tools supporting symbol-level conceptual models are concerned with rules, objects and other symbol-level entities (Newell, 1982). Tools adopting method-based conceptual models present a model of a particular problem-solving method—for example, methods for classification, planning or synthesis. Tools providing task-based conceptual models are tailored to a specific task in a particular domain.

The tradeoff between general tools that can be used in a broad variety of situations and highly supportive, specific tools is a classical software-engineering problem that applies to KA tools as well as to conventional computer programs. (In fact, this tradeoff is a general engineering problem too.) Meta-tools are designed to provide the means to escape from this dilemma by making it easy to produce and custom-tailor new KA tools. As depicted in Figure 1, knowledge engineers can use meta-tools to create new KA tools suited to their particular needs; the KA tools in turn are used by experts to create knowledge bases. In particular, meta-tools are useful for the development of

*On leave from the Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden

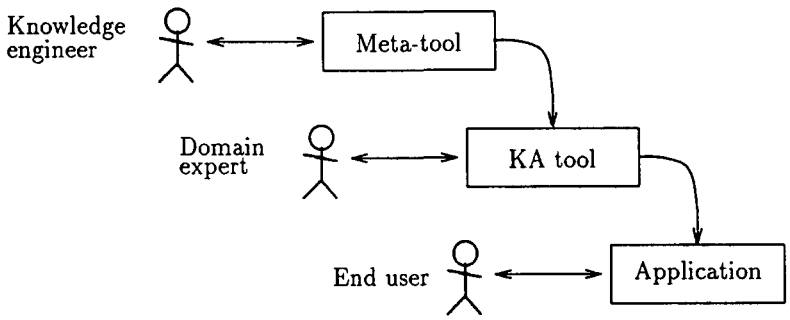

Figure 1 The fundamental basis of meta-tools. The knowledge engineer uses a meta-tool to develop a specialized KA tool that is used by a domain expert to enter knowledge for an application system

domain-oriented KA tools that incorporate task-based conceptual models, since the range of domain-specific models is large.

As we shall see, the use of meta-tools uncovers a different set of problems at the metalevel, which is the tool-design and KA-model level, because the task of a KA tool is quite different from the task of the application system. There are many ways in which a target KA tool can be described, and there also are many ways in which KA tools can be specified (i.e., described in such detail that they can be implemented automatically). This article discusses a number of such *meta-views* (or conceptual models for meta-tools), and describes the strengths and weaknesses of each. Different implementations of meta-tools are presented according to the meta-views that the meta-tools support, and the meta-tools are categorized according to their meta-views.

This article is structured as follows. Section 2 provides the background in terms of conceptual models of software. Section 3 describes several meta-views. Section 4 compares these meta-views and discusses approaches to combine and improve them. Finally, a summary and conclusions are given in section 5.

2 Background

The purpose of this section is to provide the general ideas behind automated generation of KA tools and to summarize different conceptual models for KA tools.

2.1 Automatic generation of KA tools

The major impediment for designers of specialized KA tools, including those adopting task-based conceptual models, is the problem of implementing such tools for a broad variety of domains at a reasonable cost. In principle, the problem can be approached in several ways. One approach is to try to find the 'right' level of generality for KA tools (i.e. to balance the cost of implementation and the level of support supplied by the tool). Note that it is not clear that such an optimal level exists. Even if such a level was proposed, it might be difficult for tool users—experts and knowledge engineers—to agree on it.

A second approach is to develop *generic* KA tools that can be configured for specific needs—for instance, for each domain. Examples of such configurations are specification of various tool properties in resource files, and wiring of subtools in the overall KA tool. These techniques form a knowledge-acquisition workbench. Simple configurations of generic KA tools can be performed by the domain expert (e.g., changing user preferences), whereas more sophisticated configurations can be performed by only the knowledge engineer (e.g., editing of resource databases).

A third approach, which is discussed in this paper, is to introduce an additional layer of tool support that is used to build specialized KA tools. In this approach, knowledge engineers use supportive environments, or *meta-tools*, to develop the KA tools that are to be used by the domain

experts. Meta-tools have the advantage of providing more generality than can generic KA tools, since knowledge engineers *specify* target KA tools, rather than merely *parameterize* generic tools. For instance, many conventional programs allow their users to custom-tailor certain predefined aspects of the program behaviour. Resource editors can be used to custom-tailor forms, menus, accelerator keys, colours, and so on. Such resource editors, however, cannot make more radical changes to programs (e.g., they cannot turn a document editor into a spreadsheet program, or *vice versa*). Hence, the range of options for custom-tailoring is limited for conventional programs, as well as for KA tools. Sometimes, there is no clear distinction between generic KA tools and meta-tool systems. Indeed, a sophisticated generic KA tool could provide the same flexibility and support as do the KA tools produced by a meta-tool.

2.2 Conceptual models of software

Most interactive programs are designed to support a specific metaphor. For example, spreadsheet programs such as Excel provide a conceptual model of cells, rows and columns on the screen. The desk metaphor is common among several window systems. Some computer games (and, more recently, work in virtual reality) even aim at creating an illusion of an imaginary world. Software developers often design interactive programs with a conceptual model in mind for the systems. In turn, end users build themselves mental models of the programs with which they work (either by training formally or by practicing). If a user's mental model of the program is reasonably consistent with the conceptual model implemented, the user will be able to understand and use the system. The more precisely a user's mental model maps onto to the actual program, the more likely she is to make full use of the program and to handle confusing or unexpected situations. (For instance, programmers can appreciate that open files cannot be deleted, whereas naive users may not understand why documents in the computer sometimes cannot be trashed.)

2.3 Conceptual models of KA tools

Interactive KA tools must adopt a conceptual model that allows their users to communicate with the tools, and to provide the relevant domain knowledge. The conceptual model forms the language in which the tool and its users communicate. Following Musen (1989b), we identify three major conceptual model types for KA tools:

1. *Symbol-level conceptual models*: comprise individual knowledge-base entities, such as rules, frames and parameters. Thus, this type of conceptual model addresses the most detailed level of the knowledge-base structure. Most commercial expert-system shells present the contents of the knowledge base in terms of such symbol-level entities (Newell, 1982). Often, these shells provide editing facilities according to the same conceptual model. Certain KA tools allow their users to edit and debug knowledge bases at the symbol level—for example, in terms of individual rules. TEIRESIAS (Davis, 1979) was an early project to explore tool support for knowledge-base refinement at the symbol level.

   The obvious drawback of symbol-level models is that tool users (e.g., domain experts) can be forced to think of their knowledge in a form that might be unnatural, since many symbol-level entities in reality are implementation details. For instance, although the medium of interaction in TEIRESIAS was a subset of natural language, users were restricted to thinking of their classification knowledge in terms of rule clauses and parameter values.
2. *Method-based conceptual models*: a more abstract way to view knowledge bases is to focus on the *behaviours* that are to be achieved by the target system. Method-based conceptual models attempt to describe particular problem-solving methods and classes of problem-solving methods.[1] For instance, Clancey (1985) has described a model for heuristic classification where

[1]Note that different terminology is used by different groups to denote problem-solving methods. Chandrasekaran (1986) uses the term *generic tasks*, whereas the term *interpretation models* is used by the KADS group (Wielinga et al., 1992). Karbach et al. (1990) discuss the terminological problem.

knowledge engineers can use terms and relationships of the model to describe the problem-solving behaviour of the system.

Models of problem-solving methods, such as heuristic classification, can be used as a basis for interactive KA tools. In such KA tools, the user dialogue is not based on the rules, frames or other low-level entities in the knowledge base; instead, terms and relationships in the problem-solving method are used as a basis for communication. Examples of such KA tools include ROGET (Bennett, 1985), MORE (Kahn et al., 1985), MOLE (Eshelman et al., 1987), and SALT (Marcus & McDermott, 1989). There are also tools that do not reveal the terms and relationships of problem-solving model to their users. Thus, users are not required to understand the model. Examples of KA tools that adopt an *implicit* model of problem solving are ETS (Boose, 1985) and its successor AQUINAS (Boose & Bradshaw, 1987).

3. *Task-based conceptual models*: in addition to tools that adopt models of problem-solving, there are also KA tools that attempt to incorporate models that reflect the *task* to be performed. (We use the term *task* to refer to the assignment and role of the application system in the organization. We use the term *domain* to refer to a real-world discipline that several application systems can address.) Generally, these KA tools provide a conceptual model of a general task that can be instantiated to particular task instances, thus defining individual knowledge bases. Such KA tools are more or less domain-oriented and, hence, typically are restricted to a narrow class of similar applications.

   An example of such a task-based KA tool is OPAL (Musen et al., 1987). OPAL is a graphical tool that allows cancer specialists to enter cancer-therapy plans for an expert system called ONCOCIN (Tu et al., 1989). Unlike method-oriented KA tools, OPAL adopts a conceptual model that includes domain-specific concepts such as *chemotherapy*, *drug* and *toxic reaction*. The problem-solving method of skeletal-plan refinement (Tu et al., 1989), which is the underlying reasoning mechanism in ONCOCIN, is transparent in OPAL. Physicians use this abstract task model of cancer therapy for entering and reviewing their knowledge in OPAL.

   Another KA tool, P10, is designed for the same problem-solving method in the target expert system, but the domain task is different; in this case, it is planning of protein purification (Eriksson, 1992). In spite of the similar problem-solving methods used by the expert systems generated, the domain terms and relationship used in the interaction differ substantially between the KA tools. Other examples of KA tools with task-based conceptual models are Student (Gale, 1987) and early versions of KNACK (Klinker et al., 1987).

Note that these conceptual models are largely from the perspective of the tool user. As we shall see, there are other types of models of KA tools as well.

In addition to conceptual models for KA tools, a conceptual model for the *domain* can be defined. Such *conceptual domain models* describe the expert's view of the domain and the relevant domain knowledge, including the latter's structure, rather than describing the KA tools.

3 Meta-views

The task of a meta-tool is to transform specifications provided by its users (typically knowledge engineers) into a target KA tool that can be used by domain experts. The conceptual model that a meta-tool presents to its users is the *meta-view* for the tool. Hence, meta-views are specification languages for KA tools. A second way of classifying meta-tools is to group them according to the type of target KA tools they produce. The knowledge-acquisition method supported by the target KA tools is an example of an aspect that is also relevant for meta-tool classification. A third way of classifying meta-tools is to describe various technical properties of the meta-tool implementations (e.g., the software environment required by the target KA tools, and incremental regeneration of target KA tools).

We shall present several meta-views and the meta-tools supporting them. Our reasons for focusing on meta-views, and for classifying meta-tools according to the meta-views that they

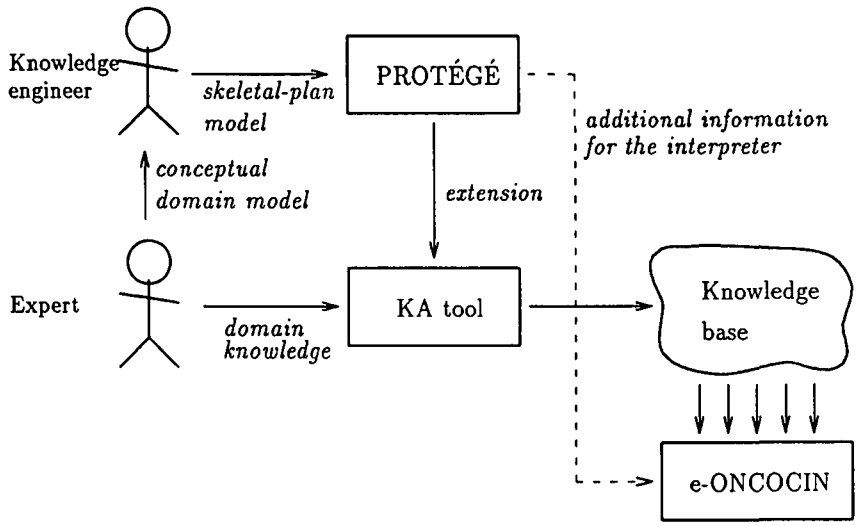

Figure 2 Generation of model-based KA tools from PROTÉGÉ. Knowledge engineers use PROTÉGÉ to instantiate model-based KA tools, which in turn are used to acquire knowledge from domain experts. The resulting knowledge bases are interpreted by e-ONCOCIN

support, are that (1) meta-views represent models of how developers think of KA tools when developing these tools, (2) meta-views shape the interaction between knowledge engineers and meta-tools, (3) meta-views show how target KA tools can be described at a level more abstract than program code, and (4) meta-views restrict the type of KA tools that can be developed with various meta-tools. From the perspective of the meta-tool user, it is important to be aware both of the meta-view supported and of that meta-view's limitations. The way in which KA tools are described and specified can differ substantially, depending on what aspects the meta-tool developer chooses to emphasize. In general, various approaches to meta-tool support can be distinguished by the meta-view adopted. Several prerequisites for meta-views can be identified. For instance, meta-views should be understandable, clearly defined, practicable and easily maintained. Note, however, that requirements for meta-views are *not* the same as requirements for KA tools or for conceptual models for KA tools. Sections 3.1–3.6 describe several meta-views that have been implemented in meta-tools, as well as other potential meta-views.

3.1 Method-oriented view

The idea on which the *method-oriented* view is based is to use a high-level description of the domain knowledge required by the problem-solving method of the target expert system as a specification paradigm for generated KA tools. Target KA tools are assumed to be domain specific and to adopt a task-based conceptual model.

3.1.1 PROTÉGÉ

PROTÉGÉ (Musen, 1989a,c) is a meta-tool that adopts a method-oriented view. The tool presents an abstract model of skeletal planning—the framework for the specification—to its users. The knowledge engineers specify properties of various entities in the planning model. As depicted in Figure 2, PROTÉGÉ transforms these specifications into a target KA tool (and also into instructions needed by the inference mechanism, e-ONCOCIN). PROTÉGÉ has an *a priori* design for the target KA tools, which resemble OPAL (Musen et al., 1987).

PROTÉGÉ assumes *skeletal-plan refinement* (Tu et al., 1989) as the problem-solving method for the target KA tools and the target expert systems. Thus, PROTÉGÉ can be seen as an

instantiation of a meta-tool for a single problem-solving method. Knowledge engineers who use PROTÉGÉ must learn the terms and relationships in the model of skeletal-plan refinement. PROTÉGÉ also assumes that the knowledge bases generated will be run by the e-ONCOCIN problem solver (Musen, 1989c), a domain-independent version of ONCOCIN. Thus, the semantics of the resulting knowledge bases are defined by the e-ONCOCIN interpreter, which uses information both from PROTÉGÉ and from structures entered and subsequently generated by a target KA tool. We shall describe briefly the method-oriented view in PROTÉGÉ because it exemplifies a practical meta-view. The model of skeletal-plan refinement comprises four components:

- *Planning entities*: are *processes* that take place over finite periods of time, such as drug treatment (as in ONCOCIN's domain). In the skeletal planning model, the planning entities are structured hierarchically. In OPAL, for example, *chemotherapy* and *radiotherapy* are considered to be potential components of *protocol* (which is the entire plan). Knowledge engineers use PROTÉGÉ to define planning-entity *classes* and—in the generated KA tools—domain experts create and edit *instances* of such planning entities. The specification of planning entities also includes *attributes* of entities in a class and various *properties* of the attributes (e.g., data type, means for establishing values and variations in values over time).
- *Task-level actions*: are unitary *operations* that control the planning entities. For example, in OPAL, task-level actions include delaying chemotherapy, attenuating the dose of a drug and ordering a laboratory test. The actions modify the instantiation of the plan at run time (i.e., they can start and stop different components of the plan and change attribute values of planning entities). Each task-level action consists of an abstract series of operations, which must be instantiated before the action can take effect. These operations are defined according to a simple script language that allows definition of the semantics of each task-level action in a domain-independent manner.
- *Input data*: can be divided into a number of user-specified data groups—for instance, data related to chemistry, to toxicity and to disease status. The data groups make it easier to handle many data items in the target KA tool, and make it possible to display only the relevant data on the workstation screen. The model of the data items includes properties for each data item, such as what are the name, prompt message, upper and lower bounds if numerical and whether the data item varies over time. In particular, every data item has associated with it a *data type* (discussed next). The target KA tool can take advantage of the type specification in creating graphical forms for data entry. For instance, if all data items in a data class have an enumerated type, the target KA tool can list in its graphical forms all the values possible.
- *Data types*: every data item is associated with data type. Data types are also needed for planning-entity attributes and for action attributes. There are a few predefined data types—namely, *boolean*, *integer*, *real* and *percent*. The knowledge engineer can introduce additional data types by specifying the input device and prompt message. For *enumerated* data types, the knowledge engineer can define *possible values* and *prompt texts* for each value.

In PROTÉGÉ, there also are facilities to enter additional low-level specifications such as *methods* and *rules*. The PROTÉGÉ model is an example of a meta-view based on a specific problem-solving method. Presumably, similar models—and meta-level tools—could be built for other problem-solving methods, such as diagnosis, troubleshooting and configuration.

To build a KA tool using PROTÉGÉ, a knowledge engineer must (1) analyse the problem area with respect to the available problem-solving method (namely, skeletal-plan refinement), (2) identify relevant *planning entities* (processes) for the domain in question, and enter them into PROTÉGÉ, (3) identify planning operations, and describe them as *task-level actions* in PROTÉGÉ, and (4) determine the input data (available at run time) for the planning process, and define *input data* and *data types* in PROTÉGÉ. If necessary, this process can be iterated as experience with the generated KA tool allows developers to clarify deficiencies in their initial model of the problem area.

3.1.2 Meta-view of PROTÉGÉ

In the method-oriented view, knowledge engineers instantiate an abstract model of the problem-solving method by specifying its properties. Through specification of particular properties of the problem-solving method, as well as of domain-related concepts used by the problem solver, the required knowledge-base structure is described sufficiently to allow generation of a KA tool for acquiring that knowledge. For example, a method-oriented view for a planning method could be instantiated by provision of domain-specific information about operations (e.g., name and ramifications), constraints and parameters (i.e., data required for planning).

Further, knowledge engineers should be able to work at an abstract level in specifying the method—that is, they should *not* have to descend to symbol-level specifications and conventional programming. (However, they might have to do so anyway, whenever they must enter knowledge that is not definable in terms of the underlying problem-solving method. Indeed, PROTÉGÉ *requires* entry of symbol-level rules for domain-specific control knowledge.) It is the task of the meta-tool to transform these high-level specifications into a working KA tool. Typically, a method-oriented meta-tool uses an *a priori* design of the target KA tool intended in this transformation; thus, the general structure of the tool is predetermined by the meta-tool developer and by the knowledge engineer. The knowledge engineer can only extend the *a priori* design by specifying various domain properties.

Note that this specification style is dependent on the actual problem-solving method used to solve a particular problem in the domain. If the problem-solving method is changed, the meta-tool must be modified accordingly. Notably, the method-oriented view (1) can be made understandable if the method supported is understandable, (2) is clearly defined if the meta-view follows a well-defined method, (3) can be made practicable (as demonstrated by PROTÉGÉ), although there may be a large gap to bridge between the specification and the target KA tool, and (4) promotes specification and maintenance, provided that the method can be modularized and that the problem can be solved by the method supported. McDermott (1988) argues that the use of explicit problem-solving methods makes expert systems easier to maintain in general.

In a way, the method-oriented view corresponds to the method-based conceptual model for KA tools (see also section 2.3). The principal difference is that KA tools that adopt method-based conceptual models often are designed to interact with *experts* in terms of problem-solving methods, whereas meta-tools that adopt the method-oriented view are intended to interact with *knowledge engineers* in terms of problem-solving methods for the purpose of specifying KA tools. Thus, another way to view a method-oriented meta-tool and the KA tools that it generates is to see them as two-level KA systems with both method-based and task-based conceptual models.

3.2 Abstract-architecture view

A fundamentally different approach is to focus on the design of the target KA tools, instead of on properties of the problem-solving method. The general idea is to use the meta-level of knowledge-acquisition *tools*—rather than the meta-level of problem-solving—for specification, since meta-tools produce KA tools, not expert systems for a domain. Thus, the *abstract-architecture* view involves a model of the desired KA tool architecture on an abstract level. Note that when we speak of *architectures* in this context, we mean architectures for KA tools, rather than architectures for knowledge bases or for target expert systems.

3.2.1 DOTS

DOTS (Eriksson, 1991b) is an example of a meta-tool supporting the abstract-architecture view. The particular meta-view adopted by DOTS is intended to give knowledge engineers as much freedom as possible in the specification. At the same time, the meta-view is sufficiently restricted to allow fast and well-defined specification. DOTS is designed to support development of KA tools for a generic class of KA methods, i.e., methods for knowledge acquisition via graphical knowledge editing by domain experts.

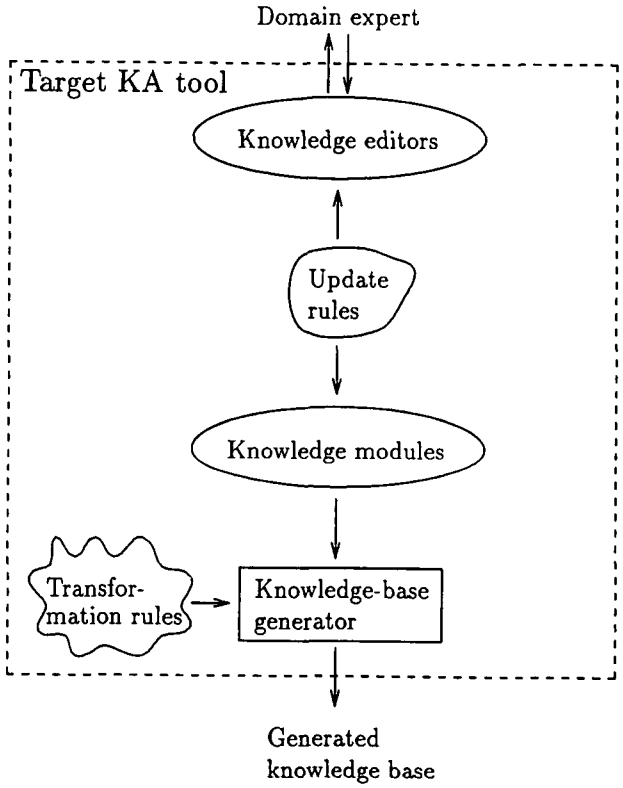

Figure 3 Components of the abstract-architecture meta-view in DOTS. The *knowledge editors* handle the interaction with the user (e.g., a domain expert), the *knowledge modules* represent the internal knowledge structures (e.g., knowledge acquired) within the KA tool, the *update rules* handle data transfer between knowledge editors and knowledge modules, and the *transformation rules* guide the knowledge-base generator in the transformation of the knowledge modules into a target knowledge base. (The dashed frame indicates the perimeter of the target KA tool.)

DOTS presents an abstract model of a KA tool. Knowledge engineers specify properties of components in the target KA tool. Such components include *knowledge editors* (which are targeted to specific knowledge chunks), *knowledge modules* and *transformation rules* for knowledge-base generation. DOTS generates a KA tool from these specifications. The major components in the DOTS meta-view include the following (see also Figure 3):

- *Knowledge editors*: the knowledge editors are components of the user interface of the KA tool under specification. Each knowledge editor is intended to edit a particular chunk of knowledge in the conceptual domain model. In DOTS, the knowledge editors of the KA tool under construction are organized in a hierarchy with a number of intrinsic editor types at the top and user-defined specializations farther down. The notion of knowledge-editor types (classes) allows several instances of a knowledge-editor to be present on the screen in the target KA tool at run time. The knowledge engineer can specify properties—such as menu layouts, window size, screen positions and icons—for each editor type and instance in the DOTS-generated tool.
- *Knowledge modules*: are used to specify the internal knowledge representation within the target KA tool. Like the knowledge editors, the knowledge modules are organized in a hierarchy with intrinsic module classes at the top and user-defined specializations below them. The knowledge engineer can specify user-defined slots for storage of knowledge acquired and of intermediate results.
- *Update rules*: the purpose of the update rules is to define the relationships between knowledge editors and knowledge modules. The set of update rules can be seen as a definition of the internal logistics structure in the target KA tool. At run time, target KA tools use update rules to guide data transfer between instances of knowledge editors and knowledge modules.

- *Transformation rules*: a set of transformation rules is used describe the generation of knowledge bases from the internal structures (i.e., the knowledge modules) in the target KA tool. The preconditions of these rules are adapted to the contents of particular knowledge modules in the KA tool, whereas the conclusions describe the knowledge-base structure to be generated. Knowledge-base generation through transformation rules is described in detail elsewhere (Eriksson, 1991a).

Figure 3 shows what the relationships are among the architectural components in the abstract-architecture view, and how they form a model of the target KA tool. At the top we find knowledge editors that handle interaction with the expert. The knowledge entered in these editors is communicated to the internal representation—the knowledge modules—through a set of update rules. Finally, a knowledge-base generator guided by transformation rules produces the knowledge base. Note that DOTS does not address the issue of providing problem-solving methods for the target expert systems. In other words, DOTS is unaware of the semantics of the target expert system—the transformation rules provide the necessary denotational semantics.

To construct a KA tool with DOTS, the knowledge engineer must (1) analyse the problem area and determine the KA tool support required, (2) define the surface structure of the KA tool in terms of knowledge editors in DOTS, (3) declare to DOTS the internal knowledge structure of the target KA tool in an object-oriented manner (i.e., as knowledge modules), (4) describe to DOTS the relationship between knowledge editors and knowledge modules in terms of update rules, and (5) define transformation rules for knowledge-base generation in DOTS. The tool-specification process does not necessarily have to proceed in this order. The ordering of steps 2–5 might be different; for instance, the internal knowledge structure of the KA tool might be defined before the user interface.

DOTS features a graphical user interface for definition of these architectural components. Figures 4 and 5 show sample screens from a session with DOTS. Several browsers give access to defined components and allow definition of new ones (see Figure 4). Since knowledge editors and knowledge modules are structured hierarchically, graph browsers are provided for them. Various tools are used to edit individual components (e.g., form-based knowledge editors; see Figure 5).

A salient aspect of DOTS is that the meta-view adopted can be modified or, if required, replaced completely. In fact, DOTS itself is implemented in DOTS according to the abstract-architecture view. Thus, DOTS is bootstrapped, which means that developers can use the system to modify and extend itself (e.g., to experiment with new meta-views).

3.2.2 SIS

Kawaguchi et al. (1991) report on a Shell for Interview Systems (SIS) that can be used to create interview-oriented KA tools. SIS represents an instantiation of the abstract-architecture approach for another class of KA tools and for another generic KA method: interviews via textual question-and-answer dialogue. SIS serves as an interview skeleton-system that helps knowledge engineers to develop interview-based KA tools such as MORE (Kahn et al., 1985), an interview-oriented KA tool for diagnostic expert systems. (Indeed, SIS has been used to recreate MORE). Interview systems generated by SIS start by requesting initial information for the construction of a basic domain model in the form of a concept network. Sentences from the expert are parsed, and are used to generate a set of task-specific *attentions* that guides the questioning in the subsequent interview. The basic domain model is refined further throughout the interview.

The meta-view adopted by SIS is basically an abstract-architecture view. Because SIS is designed to generate interview-based KA tools, the particular architecture organization is different from meta-tools that generate graphical knowledge editors (such as DOTS). Nevertheless, the SIS model incorporates architectural components common to interview-oriented KA tools. In the SIS model, the architecture of the target KA tool is divided into three layers: (1) an *interview-system* layer that provides an overview of the system components; (2) an *internal-structure* layer that describes the system in terms of program modules; and (3) a *knowledge* layer that describes the

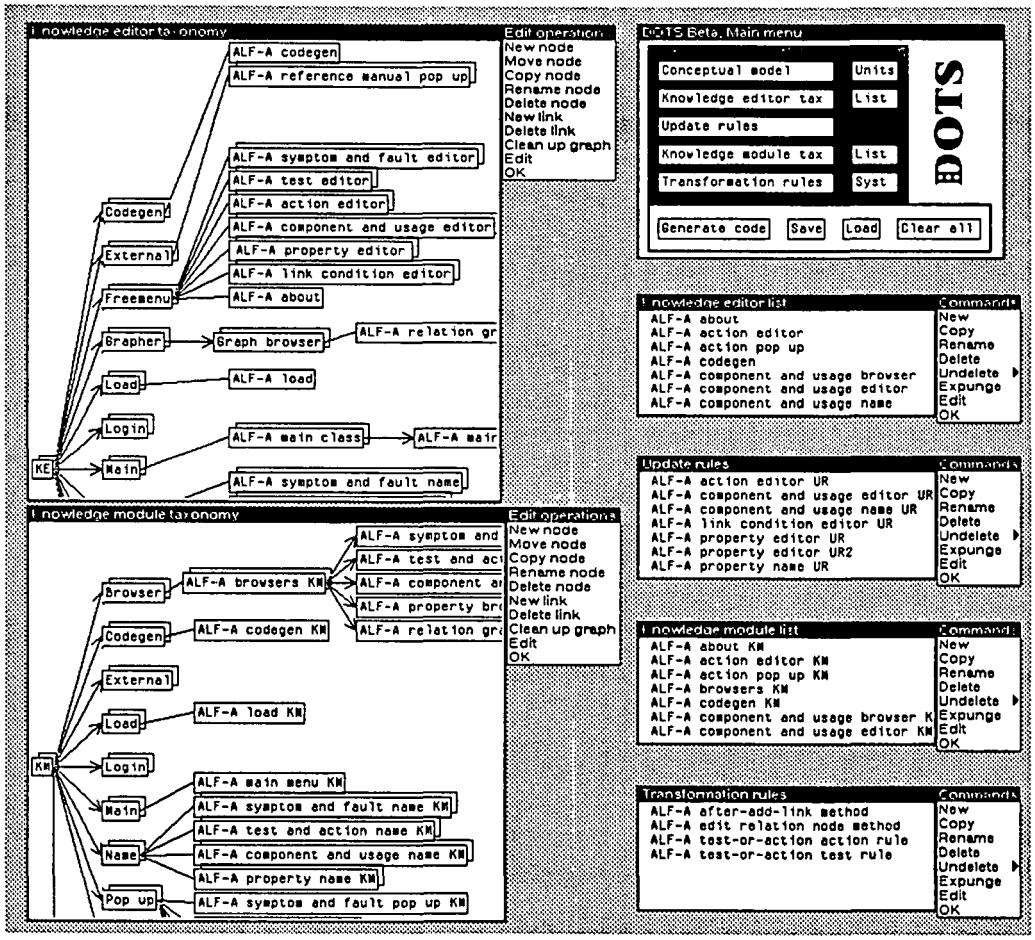

Figure 4 A sample screen from DOTS. Several different browsers are used to access architectural components defined in DOTS

behaviour of each module in the internal-structure layer. This hierarchical structure reflects the design of implemented KA tools. The most detailed specification level, the knowledge layer, comprises (1) a domain ontology, (2) a set of user-defined *generic attentions* (which can be composed from primitive attentions), (3) a specification on when to apply certain generic attentions, (4) a procedural description that controls the interview flow, and (5) a set of specific query texts. Generic attentions are a central part of the interview systems that SIS generates, since the questioning is driven by the generation and processing of attentions.

To create a KA tool with SIS, a knowledge engineer must analyse the interview situation, to model the target KA tool on each of the three abstraction layers, and to provide a detailed specification of the knowledge layer (including items 1–5). Although both SIS and DOTS adopt the abstract-architecture view, they are different meta-tools from the user's point-of-view. They use different sets of primitives for describing the architectures of the target KA tools. SIS and DOTS provide the architectural components required for developing interview systems and graphical knowledge editors, respectively. Further, these meta-tools cannot be used to generate target KA tools outside the scope of their toolboxes; for instance, DOTS cannot generate interview systems, and SIS cannot generate graphical knowledge editors. In fact, SIS has been used to regenerate MORE, and DOTS has the potential of generating KA tools such as OPAL. These facts illustrate that the abstract-architecture view is dependent on the class of target KA tools, and *vice versa*.

3.2.3 Meta-view of DOTS and SIS

In the abstract-architecture approach, knowledge engineers specify target KA tools according to an abstract model of the latter's architecture: knowledge engineers specify properties of components and component behaviour in the target KA tool architecture, rather than properties of the

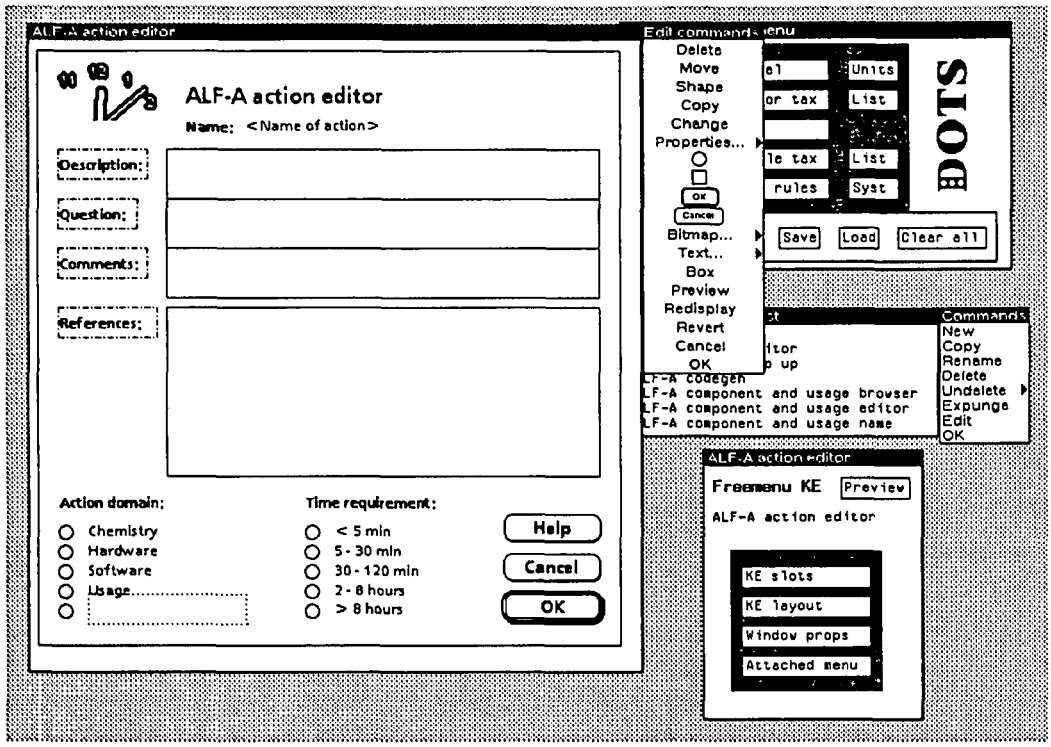

Figure 5 The layout editor in DOTS. Knowledge engineers can edit layouts for form-oriented knowledge editors graphically

domain in question with respect to the problem-solving method. Hence, we use the term *abstract-architecture view* for this specification strategy.

The task of the meta-tool is to transform the abstract specification of the architecture into an efficient implementation of the target KA tool. Note that the KA tools generated do not necessarily have to be implemented according to the architectural framework used for specification. (In principle, it is possible to conceive a meta-tool that produces target KA tools that are fundamentally different in their implementation from their abstract-architecture specification.) The abstract architecture is actually a *specification language* for KA tools. Unlike tools such as PROTÉGÉ, meta-tools supporting the abstract-architecture view have no *a priori* design for the target KA tools; instead, it is the knowledge engineer's task (and freedom) to design an appropriate target KA tool. Through specification of KA tools according to the abstract-architecture view, developers can fashion meta-tools independent of the problem-solving method. (For example, transformation rules can be used to implement flexible knowledge-base generation for a variety of target environments.) However, meta-tools supporting the abstract-architecture view are restricted to the type of architecture supported.

Notably, the abstract-architecture view (1) requires training for knowledge engineers (but this training allows them to specify KA tools applicable to many problem-solving methods); (2) has a suitable scope, since the meta-view is not limited to a particular method (it can be difficult, however, to anticipate the exact limitation profile); (3) is practicable (as demonstrated by DOTS and SIS), since there is a relatively small gap between the specification and the target KA tool; and (4) promotes specification and maintenance, since it can be modularized, and since it is not limited by a problem-solving method. The abstract-architecture view usually requires a thorough analysis of the KA tool support needed before specification of the KA tool is commenced. In this respect, the abstract-architecture view is similar to symbol-level conceptual models for KA tools. However, meta-tools are generally intended for knowledge engineers and not for experts.

Unlike the method-oriented view, the abstract-architecture view does not have a direct correspondence in a conceptual model for KA tools. The abstract-architecture view is an approach

to configuring KA tools from a library of reusable components. Just as common modules can be used for compiler construction—for instance, scanners, parsers, symbol tables and code generators—modules such as knowledge editors, various knowledge representations and knowledge-base generators can be identified for KA tools.

3.3 Organizational view

The idea behind the *organizational* view is to identify the role of the target expert system in an organization (e.g., a business organization) and to use this role as a starting point for specification of the problem solver and a corresponding KA tool (Klinker et al., 1991). The meta-tool is assumed to provide a library of typical organizational structures. Users (knowledge engineers or application experts) are asked to identify the relevant position and role in the organization.

Work at Digital Equipment Corporation (DEC) has explored the organizational view. Klinker et al. (1991) describe an architecture (Spark, Burn and FireFighter) that adopts a role-limiting approach with respect to an organization. The Spark, Burn and FireFighter model is actually a lifecycle view that takes into account the entire development process. One of the first phases of this development model is *task analysis,* in which the task of the application system is identified and described. The task analysis includes an organizational model for identifying the system's role in the organization. In other words, the organizational view is one of the models in the Spark, Burn and FireFighter approach.

The Spark, Burn and FireFighter approach emphasizes usable, as well as reusable program components as an instrument for developing expert systems. Although the goal is to support configuration of a large class of software, domain-specific KA tool support also is an integral part of this approach. Spark provides the developer with a model of activities in an organization—the *enterprise model.* Spark asks the developer to identify the task in terms of the industry. The type of industry determines the vocabulary of the subsequent dialogue. The first step in developing a problem solver with Spark is to identify the type of organization (e.g., discrete manufacturing, process manufacturing, service industries and government). The second step is to identify the specific type of industry (e.g., automotive, clothing and electronics). Different activities within the specific type of organization are presented as *activity diagrams* to the developer. The activity diagrams can be seen as semantic networks that describe activities such as selling and distributing (see Figure 6). The developer may specialize such activity diagrams for the domain in question. Spark uses activity diagrams to look up problem-solving methods (or mechanisms) that can automate various tasks in the organization. Often, there is more than one mechanism or combination of mechanisms that can be used to solve the problem. In such cases, Spark resolves the ambiguity by asking questions to differentiate between candidate mechanisms. Spark also asks questions to ensure that the information required by the mechanisms is indeed available from either the developer (the expert) or the end user.

The task of Burn is to elicit task expertise from the developer. Burn uses a library of KA tools for acquiring the knowledge required by mechanisms that constitute the problem solver. Each KA tool in the library is associated with a mechanism. In fact, Burn can be seen as a form of scheduler, since it invokes appropriate KA tools for mechanisms configured by Spark. FireFighter debugs application programs developed by Spark and Burn. FireFighter addresses two basic types of failures in application programs: (1) insufficient or inaccurate expertise for the mechanisms, and (2) inappropriate mechanisms for the task. FireFighter attempts to detect missing expertise and inappropriate mechanisms by invoking mechanism-specific diagnostic code.

The original role-limiting approach was modified subsequently in favour of smaller and decomposable mechanisms that define knowledge roles (Marques et al., 1991). The reason for change was mainly that the organizational perspective did not provide sufficient information for automatic program generation. Furthermore, the original role-limiting approach did suffer from a static model of the organization. The group at DEC is now considering a new model that will relax some of the assumptions by making task analysis an integral part of the overall framework. This

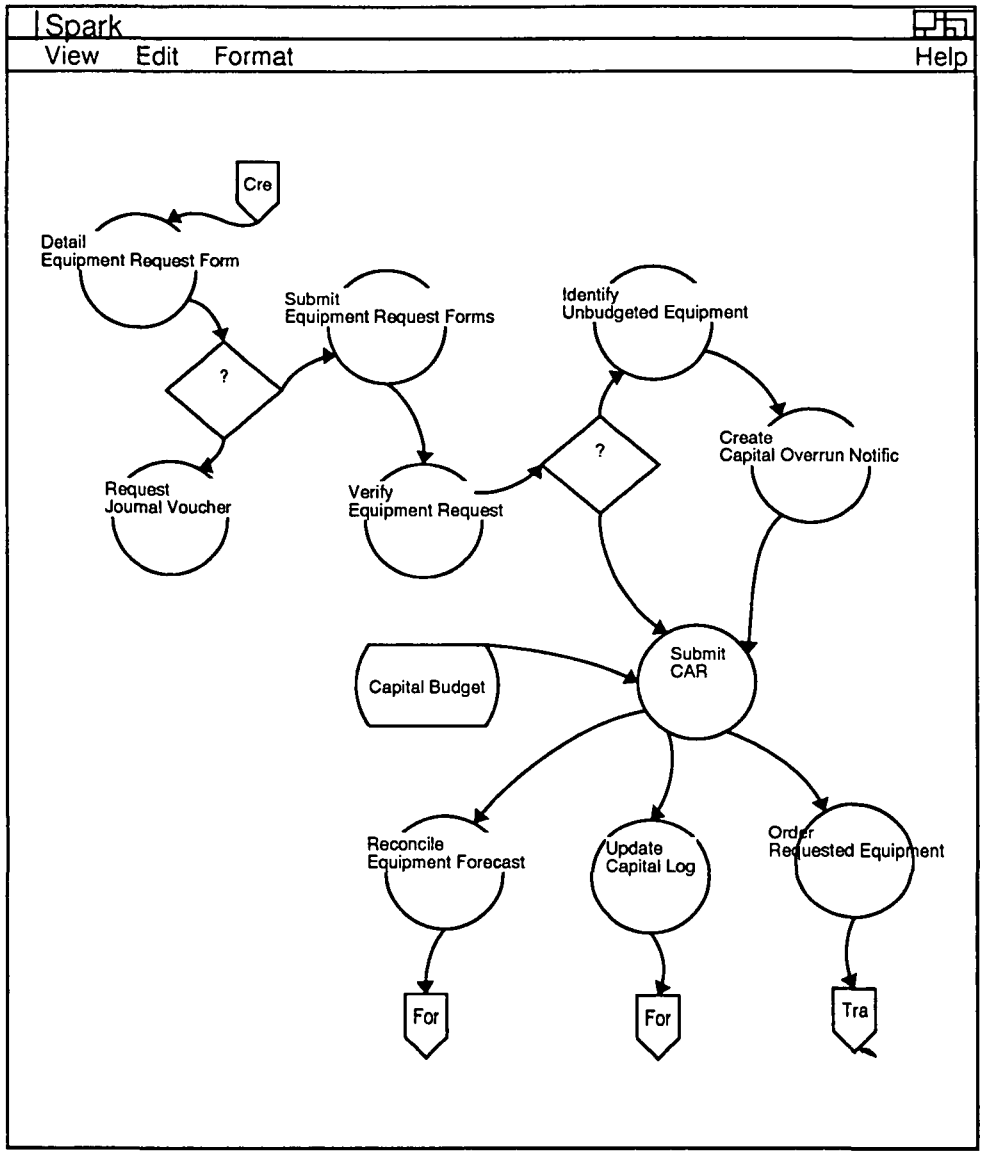

Figure 6 A sample activity diagram from Spark. This diagram describes the 'Place Order' function, which is part of a capital equipment management application. *Source:* Georg Klinker, Digital Equipment Corporation (used with permission)

approach allows for a dynamic model, but introduces the problem of how to map from a task analysis to mechanisms.

The KADS approach includes an organizational model that is used to identify the role of the expert system in order to state the requirements of the system (Wielinga et al., 1992). This description includes functions, tasks and bottlenecks in the organization, as well as a prediction of how the introduction of the system will influence the organization. Like the Spark, Burn, and FireFighter approaches, KADS is really a lifecycle model for development of expert systems. The organizational model is only one of the many models in KADS. KADS also includes *application models* that define the function of the system in an organization; *task models* that describe how the function of the system is achieved through a number of tasks the system will perform; *models of cooperation* that assign tasks to agents and specify cooperation among subtasks; *models of expertise* that describe the problem-solving expertise required to perform the problem-solving tasks; *conceptual models* that are abstract descriptions of objects and operations that the system should incorporate, expressed an implementation-independent way; and *design models* that reflect the

actual system design. However, the models of KADS are primarily intended to form an overall development methodology; they are not intended as meta-views in meta-tools. The creation of domain-specific KA tools by meta-tools has never been an explicit goal of KADS.

3.4 Ontology view

The *ontology* view is based on the idea that a description of concepts and their relationships in a domain (especially those that should be acquired by the target KA tool) can be used for specifying a KA tool that will be used to elicit detailed information about such concepts. The domain concepts used for specification are assumed to be relatively general; they are sufficiently specific to allow for automatic KA tool generation, but are sufficiently general to provide flexibility in the KA tool generated. For example, in the domain of troubleshooting automobile breakdowns, relatively general concepts—such as *electrical system*, *motor*, *spare part* and *symptom*—can be used to describe what the KA tool should handle, whereas more specific concepts (and relationships)—such as *fan belt, low oil pressure* and *broken fuse*—should be entered by domain experts in the KA tool generated.

This meta-view differs from those previously mentioned in that it focuses on the *declarative structures* required in the knowledge base, rather than on the instantiation of a problem-solving method, KA tool architecture, or organization. Ontologies in the ontology view are concerned with elements relevant for the target knowledge-base implementation, whereas ontologies in meta-tools such as PROTÉGÉ and DOTS address terms and relationships relevant to problem-solving and KA tool architecture, respectively. Hence, the ontology view is knowledge-base oriented. The knowledge engineer analyses the knowledge requirements for the emerging expert system, and creates an ontology reflecting the declarative structures needed in the target knowledge base. To specify completely a target KA tool, however, the knowledge engineer must add much information to the ontology—for instance, what to acquire (specifically), form layouts and transformations to the target representation. Nevertheless, a semantic graph of domain concepts and their relationships can be a useful part of a meta-view. Barbuceanu (1990) suggests generation of KA tools (as well as inference engines and initial knowledge bases) from a conceptual taxonomy (of the domain).

3.5 Programming languages as meta-views

The simplest form of a meta-tool could be a general-purpose programming language (possibly augmented with a few suitable toolboxes). Many KA tools have been implemented successfully in Lisp, C, Prolog, and so on. In this case, the programming language itself is the meta-view; that is, it is the specification language for the KA tool. In fact, even tools such as PROTÉGÉ and DOTS require programming languages for entries outside their meta-views. In another sense, PROTÉGÉ and DOTS *are* high-level programming languages, although they are hardly Turing-complete. Some implementations of KA tools use special packages for user-interface management. HyperCard has been used in combination with programming languages—for instance, with Lisp. Evans (1990) suggests the use of HyperCard as an environment for building expert systems. HyperCard stacks are used as the user interface, whereas modules implemented in Lisp are used to guide the acquisition process and to allow knowledge-base generation. Motta et al. (1990) describe the use of domain-oriented *coding sheets* implemented in HyperCard for direct knowledge entry by domain experts.

Naturally, substantial work is required to design and code a KA tool in a general-purpose programming language. Augmenting a programming language with a toolbox that includes common functions needed in KA tools (e.g., graphics, file input–output and knowledge base generation) can reduce the coding as well as the design tasks. If a specialized language is provided for implementation of KA tools, the coding task can be reduced further. FormIKA (Bennett, 1990) is an example of such a specialized language for implementation of forms in KA tools. The

FormIKA compiler takes as input textual descriptions of forms (including a specification of buttons and fields), constraints and relevant links to databases for persistent storage of data entered in forms. Another language that provides similar functionality for graphs, rather than for forms, is under development (Puerta et al., 1991). This work is part of an ongoing program to develop a KA-tool generator for the PROTÉGÉ-II project, the successor to PROTÉGÉ.

Gappa (1991) describes a toolbox for implementation of user interfaces for KA tools. This toolbox is essentially a program library that helps developers to create the KA tool's user interface—a laborious subtask in the implementation of graphical KA tools.

3.6 Composed meta-views

An obvious question is whether two or more meta-views can be merged to avoid some of the disadvantages in each of the original meta-views. For instance, a combination of a method-oriented view, such as the one implemented in PROTÉGÉ, and an abstract-architecture view, such as the one implemented in DOTS, would render a meta-view with the structure and guidance given by a problem-solving method as well as a high degree of flexibility in custom-tailoring of the target KA tool. Such combinations of meta-views should be regarded as *new* meta-views (rather than as simple combinations), since they represent distinct conceptual models for meta-tools. For example, a combination of the method-oriented and the abstract-architecture meta-views could require a user dialogue in the meta-tool different from a straightforward merger of the user interfaces in PROTÉGÉ and DOTS, and possibly could require an alternative analysis of the domain, as well as a new way of working with the meta-tool.

There are, however, several theoretical and technical difficulties associated with implementing such a combined meta-view, since the meta-views in PROTÉGÉ and DOTS are partly incompatible. An example of a technical challenge is that changes to one part of the combined meta-view might affect other parts of the meta-view in a non trivial way. Sometimes this problem is not merely a technical one. For instance, it is sometimes possible to infer the structure of a form from an instantiation of a problem-solving method. The layout of forms generated can be custom-tailored later according to user preferences. Nevertheless, it is not clear what should be done if nontrivial modifications are made to the original instantiation of the problem-solving method. Pragmatic solutions might involve ignoring the changes in the method instantiation, discarding the custom-tailored layout or applying heuristics to change the form layout.

There are many still issues to be resolved before composed meta-views can be implemented. A principal challenge for Spark (and for PROTÉGÉ-II, the successor of PROTÉGÉ) is to generate KA tools that can seamlessly merge the knowledge requirements of all the components. Further, to achieve high-quality KA tools for composed problem solvers, the meta-tool must combine and structure a conglomeration of knowledge requirements into knowledge editors that are meaningful for experts.

4 Discussion

We divide our discussion into two parts: (1) a comparison between different approaches, and (2) an examination of implications for future meta-views and meta-tools.

4.1 Comparison of meta-views and meta-tools

At this stage, it is not possible to undertake a quantitative study comparing performance among various meta-views. We distinguish four criteria for comparing meta-views:

1. *Restrictions*: the scope, or limitations, of the meta-view. Every form of specification strategy will have boundaries in terms of the problem classes it addresses. (General programming languages are, for instance, restricted to computable problems.)

Table 1 A comparison of four different meta-views, as well as of conventional programming languages. The aspect 'Section' refers to relevant sections in this article. (KE = knowledge engineer; DE = domain expert, N/A = not available)

Aspect	Method-oriented	Abstract architecture	Organizational	Ontology	Programming language
Section	3.1	3.2	3.3	3.4	3.5
Restricted to	problem class	KA tool architecture	predefined mechanisms & KA tools	domain description language	computable problems
Supportive power	High	High	High	N/A	Low
Meta-tool user(s)	KE	KE	KE & DE	KE & DE	Programmers
Training required	Yes, problem-solving modeling	Yes, KA tool design	Yes, domain modeling	N/A	Yes, programming

2. *Supportive power*: the degree of relevant modelling support provided by the meta-view (roughly corresponding to the level of abstraction). High supportive power usually entails use of a specialized specification strategy. Just as in any language, supportive power must be balanced against restrictions in meta-views.
3. *Meta-tool users*: the intended user category for the meta-tools supporting a meta-view. To create a meta-view (and thus meta-tool), we must know which members of the development team will use it.
4. *Training*: the training required to understand and use the meta-view for KA tool modelling. For instance, are general knowledge-engineering skills sufficient, or is additional training needed for a knowledge engineer to specify useful KA tools in a specific meta-view?

Table 1 summarizes these properties for four different meta-views: the method-oriented, abstract-architecture, organizational and ontology meta-views. In addition, the corresponding properties for general programming languages are shown.

The method-oriented and abstract-architecture views have highly specialized perspectives. These meta-views are also intended for knowledge engineers trained in modelling according to the perspective adopted. To our knowledge, the ontology view has not been implemented in a meta-tool; consequently, no practical experience in this specification strategy has been reported. General programming languages provide indisputable generality, but much work is required to implement interactive KA tools in them. The developers must have programming skills and knowledge about the language, the environment, the window system, and so on.

Implementation of meta-tools can realize a meta-view in various ways. Designs (meta-views) are underspecified; therefore, examining implementations (meta-tools) makes comparisons more concrete. The same meta-view can be implemented in different manners, depending on considerations addressed by the meta-tool developer (e.g., dialog style, user community and computing environment). At this time, only a few meta-tools—or tools exploiting meta-level structures—have been reported, so it is difficult to compare different meta-tools (and their underlying meta-views), especially by their strengths and weaknesses in practical use. Nevertheless, it is possible to make a rough classification of those available and to point out areas for development of new systems. We distinguish six criteria for meta-tools:

1. *Separate meta-tools and KA tools*: a distinction can be made between approaches that separate tool support into two or more levels (e.g., meta-tools and KA tools) and systems that maintain both levels in the same tool (e.g., KA workbenches). Integrated meta-tools and KA tools can

Table 2 Comparison of five tools that have meta-components (KNACK is not a meta-tool, but rather is a KA tool that uses meta-level representation of the domain to guide interrogation processes)

Aspect	PROTÉGÉ	DOTS	SIS	Spark et al.	KNACK
Separate meta- and KA tools	Yes	Yes	Yes	Yes	No
Meta-view	Method-oriented	Abstract architecture	Abstract architecture	Organizational	N/A
Specification model	Skeletal-plan refinement	Editor-module architecture	Generic attentions	Tasks and mechanisms	N/A
Assumed domain class	Protocol management	None	None	Several domain classes	Reporting tasks
Target KA tool/method	Task-specific	Task-specific	Interview-oriented	Mechanism-specific	Interview-oriented
Target language	e-ONCOCIN	No assumptions	—	OPS5	OPS5

potentially allow greater flexibility in terms of changes on the meta-level (e.g., faster update of the KA tool specification), whereas distinct meta-tools and KA tools can potentially run in different environments on different platforms.

2. *Meta-view*: the meta-view is the specification strategy (or model of the meta-level) for KA tools adopted (see section 3).
3. *Specification model*: the specification model is the specific modelling framework used for specifying KA tools. For a method-oriented view, the specification model can be the particular model(s) supported; for the abstract-architecture view, the specification model would be the architectural framework and its components.
4. *Assumed domain class*: meta-tools may assume a problem-solving method (e.g., classification and planning) for KA tools generated. The class of KA tools generated is also constrained by the meta-view. Generality in the domain class means that a meta-tool is useful in a broad variety of domains, whereas a restricted domain class could allow easier KA tool specifications since less definition information is required.
5. *Target KA tools family*: target KA tools (or in the case of an integrated meta-level, the tools itself) might adopt different knowledge-acquisition strategies; for example, they might use explicit knowledge editing, as in PROTÉGÉ and DOTS, and interview-based knowledge elicitation (i.e., KA tools oriented toward interrogation of domain experts in restricted natural language), as in SIS.
6. *Target language*: the target inference-engine or target knowledge-base format might vary. Meta-tools may or may not assume a particular inference engine or format for the resulting knowledge base.

Table 2 summarizes these properties for five systems: PROTÉGÉ (Musen, 1989c); DOTS (Eriksson, 1991b); SIS (Kawaguchi et al., 1991); Spark, Burn and FireFighter (Klinker et al., 1991; Marques et al., 1991); and KNACK (Klinker et al., 1987).

PROTÉGÉ, DOTS and Spark are considered to be pure meta-tools in the sense that they separate the meta-level and KA level. Although KNACK features flexibility in adapting meta-level aspects of the KA tool, it was not originally intended as a meta-level tool for knowledge acquisition. As indicated in Table 2, the salient property that distinguishes the meta-tools PROTÉGÉ, DOTS and Spark is the class of meta-view that they support. These tools also use specific models, which can be viewed as instances of meta-views; for example, PROTÉGÉ assumes

skeletal-plan refinement as the method, DOTS assumes an architecture based on knowledge editors and modules, SIS relies heavily on attentions in its architectural model, and Spark is based on tasks and mechanisms for the method. PROTÉGÉ is tailored to a specific domain class (problems solvable by skeletal-plan refinement), whereas Spark supports several domain classes. Since the abstract-architecture view is independent of the domain, DOTS and SIS do not make assumptions about the domain class. In terms of target KA tool family, PROTÉGÉ, DOTS and Spark all generate domain-oriented KA tools with graphical user interfaces for knowledge editing (the tools that Spark configures are mechanism specific, however), whereas SIS and KNACK assume an interview-oriented dialog with the expert. PROTÉGÉ is designed for e-ONCOCIN as the target language for knowledge bases produced by the KA tools. KNACK produces rule bases. DOTS does not assume a particular target language; transformation rules are used to ensure insulation from the target language in DOTS. The knowledge engineer, however, is assumed to know the target language and the transformation-rule format.

4.2 Examination of implications of the comparison for meta-views and meta-tools

An advantage of the method-oriented view is that it offers simplicity in the specification; i.e., to specify KA tools, knowledge engineers need to be trained in only the problem-solving method. Another advantage is that an inference engine can be parameterized by largely the same specifications. The *a priori* design for target KA tools can be both a limitation and a strength. The abstract-architecture view is not limited to a single problem-solving method in the same way as is the method-oriented approach; its limitations can be found on the level of target KA tool architectures. However, the specification is more complex, and knowledge engineers need training in the abstract architecture supported. Since no *a priori* design for the target KA tools is presupposed, knowledge engineers are given extra freedom in custom-tailoring the tools generated.

Another important issue that concerns all types of meta-view is that of generality. Finding the 'right' level of generality for meta-tools is primarily an empirical question. The implementations discussed in section 3 are merely data points that can guide the development of the next generation of meta-tools. Meta-tools, and thus meta-views, need to be sufficiently general to be useful for a significantly large user community. Presumably, future generations of meta-tools must be generalized beyond current tools. In the method-oriented view, many different problem-solving methods can be supported by different meta-tools. Likewise, in the abstract-architecture view, many types of generic architectures can be supported. Tools such as PROTÉGÉ and DOTS adopt only specific *schemata* of the general meta-view. For instance, it is possible to conceive other method-oriented meta-tools for completely different classes of problem-solving methods.

4.3 Potential meta-views

In principle, it is possible to conceive of meta-views other than the ones we have mentioned. None of the previously described meta-views takes into account different knowledge-elicitation methods. The meta-tools incorporate *a priori* elicitation methods, such as knowledge editing and interviewing. KA tools based on elicitation methods other than the ones predefined cannot be described easily in these meta-views. It is desirable to have meta-views that allow a certain degree of freedom in the selection of elicitation methods, or alternatively, that provide a description language for elicitation methods for a particular kind of knowledge.

The method-oriented view and the abstract-architecture view are both primarily intended for target KA tools with task-based conceptual models, because the benefit of meta-tools is largest when the meta-tools are used to generate specialized KA tools. The conceptual models for KA tools mentioned in section 2.3 also include symbol-level and method-based conceptual models. Hence, there are many types of KA tools that might require different meta-views to allow generation of such KA tools from meta-tools. Consider that problem-solving methods can always

be made more specialized or more domain-specific. Therefore, any conceptual model can be a meta-view of a model that is yet more specialized. However, the advantage of using meta-tools, rather than using manual implementation, lies in development of many domain-specific tools.

Both PROTÉGÉ and DOTS use specialized editors for explicit entry and modification of the target KA tool specification. Thus, the field is open for meta-views and meta-tools that are based on *implicit* methods for acquisition of the target KA tool specifications. One approach could be to generate KA tools from *examples* of knowledge formulated in the conceptual framework that the target KA tool is supposed to adopt, and from sample fragments of the corresponding representation in the knowledge base. This strategy can be useful when KA tools are to be introduced to *existing* knowledge bases (e.g., to support knowledge-base maintenance or to renovate a knowledge base). In this approach to KA tool generation, the task of the meta-tool is to abstract a KA tool from such examples of input and required output. It might not be possible, however, to generate a complete KA tool from such examples. A more realistic approach would be to generate templates that could be used for further refinement.

4.4 Ongoing work

The ongoing PROTÉGÉ-II project will remove several of the deficiencies in the method-oriented view supported by the original PROTÉGÉ implementation (Puerta et al., 1992). The principal drawback of the method-oriented view in PROTÉGÉ is that only a single problem-solving method is supported, which means that the meta-tool is restricted to a limited class of problems. To generalize the underlying problem solver beyond a single method, PROTÉGÉ-II provides the user with a set of reusable method components that can be combined into a problem solver for the domain in question. Hence, this part of the PROTÉGÉ-II system addresses the problem of configuring a problem solver that operates on the knowledge structures obtained by the target KA tool. Another goal for PROTÉGÉ-II is to generate KA tools from method configurations. Such a configuration can provide a basis for the knowledge requirements and, therefore, for the KA-tool design. Nevertheless, the meta-tool will require additional information to specify high-quality KA tools completely, since a problem-solver specification does not intrinsically stipulate aspects of the target KA tool such as knowledge-acquisition method, user dialogue, and form layouts. Thus, the goal of PROTÉGÉ-II is to provide a combined meta-view.

5 Summary and conclusions

Meta-level KA tools can be used to generate tailored KA tools automatically from a specification. Just as KA tools present different conceptual models to their users, meta-tools provide alternative meta-views to their users. Thus, a meta-view is the specification language used for describing the target KA tool.

This separation of KA-tool specification and implementation is analogous to the distinction between *epistemological* and *heuristic* adequacy for knowledge representation (McCarthy & Hayes, 1969). There are several epistemological problems at the level of meta-views for KA tools—for instance, how to view emerging target KA tools. The task of implementing a meta-tool includes solving the problem of establishing heuristic adequacy for the meta-view (i.e., ensuring the practicability of the meta-view). In fact, epistemological and heuristic adequacy must be ensured at every system level (i.e., expert system, KA tool and meta-tool).

We have examined several different meta-views that can be used for KA tool specification. For instance, the method-oriented view adopts the metaphor of a problem-solving method, and the abstract-architecture view focuses on components of KA tools. Meta-views govern the design process of the target KA tools. Knowledge engineers might be encouraged to view the KA tool support in a certain manner, and to design their KA tools in certain ways. No meta-view (except perhaps a general programming language) is complete, in the sense that all conceivable KA tools can be specified in it. Although there are theoretical and practical difficulties in combining

(sometimes partially incompatible) meta-views, doing so can remove certain biases introduced by a single meta-view in the knowledge-engineering process, improve the generality of the specification language and provide more freedom for the knowledge engineer in designing KA tools.

Acknowledgements

This work has been supported in part by grants LM05157 and LM05208 from the National Library of Medicine, by a gift from Digital Equipment Corporation, and by scholarships from the Swedish Institute, from Fulbright Commission, and from Stanford University. Dr. Musen is recipient of National Science Foundation Young Investigator Award IRI-9257578.

We are grateful to Lyn Dupré for providing editorial assistance. John Egar and Samson Tu commented on various draft versions of this article.

References

Barbuceanu, M, 1990. "The MODELS environment for knowledge acquisition: Automating the task-specific architecture approach" In: JH Boose and BR Gaines (eds.), *Proceedings 5th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, pp 1.1–1.20.

Bennett, A, 1990. FormIKA: A form-based user interface management system for knowledge acquisition. Technical Report KSL-90-43, Knowledge Systems Laboratory, Stanford University, Stanford, CA.

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

Boose, JH, 1985. "A knowledge acquisition program for expert systems based on personal construct psychology" *International Journal of Man-Machine Studies* **23** (5) 495–525.

Boose, JH and Bradshaw, JM, 1987. "Expertise transfer and complex problems: Using AQUINAS as a knowledge-acquisition workbench for knowledge-based systems" *International Journal of Man–Machine Studies* **26** (1) 3–28.

Chandrasekaran, B, 1986. "Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design" *IEEE Expert* **1** (3) 23–30.

Clancey, WJ, 1985. "Heuristic classification" *Artificial Intelligence* **27** (3) 289–350.

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

Eriksson, H, 1991a. "Architectural issues in KA tools: Towards structured transformation into knowledge-bases" In: *Proceedings 5th European Knowledge Acquisition for Knowledge-Based Systems Workshop*, Crieff, Scotland.

Eriksson, H, 1991b. Meta-Tool Support for Knowledge Acquisition, PhD thesis, Linköping University, Sweden.

Eriksson, H, 1992. "Domain-oriented knowledge acquisition tool for protein purification planning" *Journal of Chemical Information and Computer Sciences* **32** (1) 90–95.

Eshelman, L, Ehret, D, McDermott, J and Tan, M, 1987. "MOLE: A tenacious knowledge-acquisition tool" *International Journal of Man–Machine Studies* **26** (1) 41–54.

Evans, R, 1990. "Expert systems and HyperCard" *Byte* **15** (1) 317–324.

Gale, WA, 1987. "Knowledge-based knowledge acquisition for a statistical consulting system" *International Journal of Man–Machine Studies* **26** (1) 55–64.

Gappa, U, 1991. "A tool-box for generating graphical knowledge acquisition environments" In: *Proceedings of the World Congress on Expert Systems,* Orlando, FL.

Kahn, G, Nowlan, S and McDermott, J, 1985. "Strategies for knowledge acquisition" *IEEE Transactions on Pattern Analysis and Machine Intelligence* **7** (5) 511–522.

Karbach, W, Linster, M and Voß, A, 1990. "Models, methods, roles and tasks: Many labels—one idea?" *Knowledge Acquisition* **2** (4) 279–299.

Kawaguchi, A, Motoda, H and Mizoguchi, R, 1991. "Interview-based knowledge acquisition using dynamic analysis" *IEEE Expert* **6** (5) 47–60.

Klinker, G, Bentolila, J, Genetet, S, Grimes, M and McDermott, J, 1987. "KNACK: report-driven knowledge acquisition" *International Journal of Man–Machine Studies* **26** (1) 65–79.

Klinker, G, Bhola, C, Dallemagne, G, Marques, D and McDermott, J, 1991. "Usable and reusable programming constructs" *Knowledge Acquisition* **3** (2) 117–135.

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

Marques, D, Klinker, G, Dallemagne, G, Gautier, P, McDermott, J and Tung, D, 1991. "More data on usable and reusable programming constructs" In: JH Boose and BR Gaines (eds.), *Proceedings 6th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop,* Banff, Canada, pp 14.1–14.19.

McCarthy, J and Hayes, PJ, 1969. "Some philosophical problems from the standpoint of artificial intelligence" *Machine Intelligence* **4** 463–502.

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

Motta, E, Rajan, T and Eisenstadt, M, 1990. "Knowledge acquisition as a process of model refinement" *Knowledge Acquisition* **2** (1) 21–49.

Musen, MA, 1989a. *Automated Generation of Model-Based Knowledge-Acquisition Tools,* Morgan-Kaufmann.

Musen, MA, 1989b. "Conceptual models of interactive knowledge acquisition tools" *Knowledge Acquisition* **1** (1) 73–88.

Musen, MA, 1989c. "An editor for the conceptual models of interactive knowledge acquisition tools" *International Journal of Man–Machine Studies* **31** (6) 673–698.

Musen, MA, Fagan, LM, Combs, DM and Shortliffe, EH, 1987. "Use of a domain model to drive an interactive knowledge-editing tool" *International Journal of Man–Machine Studies* **26** (1) 105–121.

Newell, A, 1982. "The knowledge level" *Artificial Intelligence* **18** (1) 87–127.

Puerta, AR, Egar, JW and Musen, MA, 1991. Automated generation of adaptable knowledge-acquisition tools with Mecano. Technical Report KSL-91-62, Knowledge Systems Laboratory, Stanford University, Stanford, CA.

Puerta, AR, Egar, JW, Tu, SW and Musen, MA, 1992. "A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools" *Knowledge Acquisition* **4** (2) 171–196.

Tu, SW, Kahn, MG, Musen, MA, Ferguson, JC, Shortliffe, EH and Fagan, LM, 1989. "Episodic skeletal-plan refinement based on temporal data" *Communications of the ACM* **32** (12) 1439–1455.

Wielinga, BJ, Schreiber, AT and Breuker, JA, 1992. "KADS: a modelling approach to knowledge engineering" *Knowledge Acquisition* **4** (1) 5–53.