N93-17508

Meta-Tools for Software Development and Knowledge Acquisition

S9-61
136883
P.6

Henrik Eriksson[*] Mark A. Musen

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

"Man is a tool-using animal.... Without tools he is nothing, with tools he is all."
Thomas Carlyle (1795–1881)

Abstract

The effectiveness of tools that provide support for software development is highly dependent on the match between the tools and their task. Knowledge-acquisition (KA) tools constitute a class of development tools targeted at knowledge-based systems. Generally, KA tools that are custom-tailored for particular application domains are more effective than are general KA tools that cover a large class of domains. The high cost of custom-tailoring KA tools manually has encouraged researchers to develop *meta-tools* for KA tools. Current research issues in meta-tools for knowledge acquisition are the specification styles, or *meta-views*, for target KA tools used, and the relationships between the specification entered in the meta-tool and other specifications for the target program under development. We examine different types of meta-views and meta-tools. Our current project is to provide meta-tools that produce KA tools from multiple specification sources—for instance, from a task analysis of the target application.

Introduction

Knowledge-acquisition (KA) tools are programs that help developers to elicit and structure domain knowledge for use in application programs (e.g., in expert systems). Typically, KA tools allow nonprogrammers who are specialists in some domain area to enter structures relevant for the application program without the aid of an intermediary who is proficient in programming. Thus, KA tools are, in a way, code-generating software-engineering tools for a restricted type of software and for a particular group of users. To increase the usability of KA tools, researchers in knowledge acquisition have experimented with specializing the tools in various ways. For instance, KA tools have been specialized to knowledge-acquisition methods, problem tasks, domains, and even applications. In most cases, specialized KA tools are reported to be more effective than general ones, because the users are nonprogrammers familiar with the domain terminology. In addition to those that involve the elicitation of knowledge from experts, there are approaches to KA tool support that rely on knowledge acquisition from texts, and there also are methods that incorporate machine learning from example solutions.

Custom-tailoring KA tools can be a laborious task. When the benefit of domain-specific KA tools is compared to the effort of developing them, the tool-development cost is often unacceptable for small projects. Also, development of domain-specific tools can in itself be a software-engineering problem. These problems have been addressed with supportive tools and, to a certain extent, with tool-development methodologies. Just as code-generator writing systems can be used to produce code-generating tools, *meta-tools* for knowledge acquisition can help developers to implement domain-specific KA tools. Several meta-tools that generate target KA tools automatically from specifications provided by the developers have been implemented by researchers in knowledge acquisition. Although KA tools are generally intended for nonprogrammers, variants of such tools can be used by programmers to increase software quality and programmer productivity. A meta-tool can be used to create the tool required by programmers.

An important aspect of a meta-tool is the specification strategy, or *meta-view*, for target tools that the meta-tool provides to the developers. The meta-view comprises the conceptual model of the target tool that the meta-tool supports, as well as the specification language for target tools. Depending on the view of target tools, several types of meta-views are possible. Domain-specific tools for software development are desirable in many situations. Meta-tools, however, preferably should be domain-independent so that they can produce domain-oriented tools for a broad area of applications.

Much of the work in meta-tool support for knowledge acquisition is relevant for software engineering, especially approaches to domain-specific development tools. If the design and implementation of such domain-oriented software-engineering tools are

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

laborious tasks, meta-level tools are certainly required. In this paper, we discuss alternative meta-views and describe their implementation in different meta-tools.

Background

Knowledge engineering and software engineering are partially overlapping disciplines. Moreover, tools for knowledge engineering and computer-aided software engineering (CASE) tools have gone through similar development stages, in the sense that increasingly specialized tools have been considered. The first-generation AI development tools were general and were essentially programming languages with integrated development environments. Examples of such tools are EMYCIN, KEE, ART, and S1. Only skilled programmers and knowledge engineers could use these tools, so the tools were inaccessible to domain specialists who had not had extensive training in computer and information sciences.

Simultaneously, investigators attempted to developed tools that acquired expertise directly from domain specialist. Initially, these KA tools were also general. In the mid-1980s, these general KA tools were followed by a second generation of KA tools that were specific to particular problem tasks—for instance, to classification, configuration, or scheduling. Even if the scope of the tool is restricted to one problem task, however, nonprogrammers may have difficulty using the tool [Marcus, 1988]. A third generation of even more specialized KA tools was therefore developed. Researchers started to experiment with domain-specific KA tools. Such tools are designed such that domain specialists can use well-known domain concepts in the tool dialog [Eriksson, 1992; Musen *et al.*, 1987].

Domain-oriented KA tools can provide effective support within their area, because they draw their power from built-in domain concepts that users can identify easily. However, the development of such KA tools is costly, since the amount of programming required to implement such domain-specific tools is large in comparison to the scope of the tools' services. There are three fundamental approaches to this problem: (1) balancing tool generality versus domain-orientation to achieve a reasonable trade-off between utility and cost, (2) improving further general tools, and (3) reducing the cost of developing domain-oriented KA tools (e.g., through technological means).

We have chosen the third approach. Our goal is, thus, to make it easier for developers to design and implement tools tailored for their needs. Meta-tools can help developers to create new domain-specific tools as well as to custom-tailor existing tools for a domain. There are two principal roles for meta-tools in this approach: (1) to address the software-engineering problem of developing (and specializing) target tools, and (2) to support the target-tool design and specification process. In addition to meta-tools, development *methodologies* that incorporate specialization of development tools can help the developer to control the development process and to reduce its cost.

One feature that distinguishes KA tools from other development tools is the intended tool user. KA tools are primarily intended for use by domain specialists, whereas code-generating software-engineering tools are generally designed to be use by developers with programming knowledge. So far, we have primarily worked with KA tools for knowledge-based systems. Nevertheless, several of our results can be generalized to other types of software-development tools.

Meta-Views

The most important aspect of a meta-tool is the specification model of the target tools that it provides to the developer. The meta-view adopted by the meta-tool guides the specification process, and determines the scope of the meta-tool. Meta-tools can differ substantially, depending on what aspects the meta-tool developer chooses to emphasize in the meta-view. Preferably, the meta-view should in some way reflect the way that developers think about the target tools, and should provide a natural way of specifying target tools. Several groups of meta-views can be identified.

The Method-Oriented View

The *method-oriented* view provides a framework for describing the problem-solving method to be used in the final application in a way that makes the description useful for generation of KA tools. Meta-tools implementing a method-oriented view produce target KA tools from a partial instantiation of a generic problem-solving method (e.g., planning, scheduling, or troubleshooting methods). Target KA tools are fully instantiated according to the expertise required by the problem-solving methods for performing their tasks. For example, the developer can instantiate a planning method by providing descriptions of *actions* (and their preconditions as well as ramifications), *constraints*, and *goals*. A domain-specific KA tool that allows specialists to enter and edit skeletal plans can be produced from such an instantiation of the planning method by a meta-tool supporting the planning method. Typically, meta-tools adopting a method-oriented view incorporate some form of a priori design of the target tools. One of the advantages of the method-oriented view is that the instantiation of a generic method structures the development process and guides the developer. Another advantage is that the target tool can be developed rapidly if a problem-solving method for the application is known.

There are, however, drawbacks of the method-oriented view. A significant problem is that the meta-tool is restricted to one particular problem-solving method. KA tools that acquire knowledge for other problem-solving methods, including KA tools for domains where the problem-solving method supported is unsuitable, cannot be specified using the method-oriented approach. Another problem is that the type of KA tools produced for a particular domain is fixed (i.e., it is possible to have only a one-to-one correspondence between an instance of a problem-solving method and its KA tool, due to the a priori KA tool design). Adapting a meta-tool for another problem-solving method is currently a laborious task that may involve a major redesign of the meta-tool.

The Abstract-Architecture View

The *abstract-architecture* view is based on an architectural model of the target tool. In this approach, the developer specifies components of the target KA tool, such as the user interface, the internal representation, and the generator for target code. In other words, to create a target KA tool, the developer has to instantiate each of the components in the KA tool architecture and to link together the components. (Naturally, this task requires a prior analysis of the domain and of the requirements on the KA tool.)

Meta-tools adopting this meta-view produce KA tool implementations from abstract specifications of target KA tool components. In a way, the abstract-architecture view is similar to specification languages found in compiler compilers (e.g., Yacc and Bison). The abstract-architecture view differs from the method-oriented view in that it focuses on the target tool rather than on the application program under development. The abstract-architecture view provides more flexibility for the developer than does the method-oriented view, because many tools potentially can be specified for one domain.

The major advantage of the abstract-architecture view is that target KA tools can be specified independently of the problem-solving method adopted. Hence, the meta-tools do *not* have to rely on specific problem-solving methods (or on any other class of domains). There are, however, other limitations: The abstract-architecture view imposes restrictions on the types of target tools that can be specified. For instance, a meta-tool supporting architectural components for graphical editing and browsing cannot easily be used to produce debugging tools (which require a completely different set of architectural components). Another disadvantage of the abstract-architecture view is that the developer needs to be aware of the architecture of the target KA tools, which knowledge is not required for the method-oriented view (where the developer is required to know only the problem-solving method).

The Organizational View

The *organizational* view captures the intended organizational context for the system under development. The idea is to derive the target system's role from an organizational model (e.g., an enterprise model) and to identify the task of the system from its role. When the task of the system has been established, it can be used together with the organizational model to specify target KA tools to a meta-tool. To specify a target KA tool according to the organizational view, a developer must (1) identify the actual organizational structure from a library of typical organizations, and (2) indicate the relevant position and role in the organization for the system. In essence, the organizational perspective is an approach to create a job description for the system.

The organizational view provides a broader perspective on KA tool specification than do the method-oriented and abstract-architecture views. The broad perspective is an advantage of the organizational view, since it helps to clarify how the system is to be used and to make this information available to the meta-tool. Another advantage of the organizational view is that organizational information is often easily available and can be provided by nonprogrammers. One of the problems with the organizational view, however, is that it is not clear whether such a model is sufficient to specify a KA tool completely. Additional information, such as identification of appropriate problem-solving methods and other technical issues, might be needed to produce automatically or semiautomatically target KA tools that can be used by people in the organization (i.e., nonprogrammers) to develop the system. A pure organizational model would not provide sufficient information, but an extended organizational model might be practical for the tool generation.

The Ontological View

The *ontological* view is based on the idea that domain concepts and relationships can be used for generation of domain-specific KA tools that incorporate such concepts and relationships. Concept definitions in the ontology can be used as a basis for automated generation of domain-oriented editors in the KA tool. Target KA tools can then be used to acquire details about the domain concepts. For example, instances of domain-specific classes in the ontology can be entered and edited in the target KA tool by developers and by domain specialists. To complete a target KA tool, however, the developer might have to provide additional information in the ontology (e.g., information about how to edit certain concepts). The ontological view differs from the previously mentioned meta-views in that it focuses on declarative structures required in the application system.

Composed Meta-Views

An important question is whether we can combine several meta-view such that we avoid some of the disadvantages of particular meta-views. For instance, a combination of the method-oriented and the abstract-architecture views can potentially render a meta-view that provides the guidance of a predetermined problem-solving method and the capability to custom-tailor the target tool (e.g., for individual users). There are, however, several conceptual and technical obstacles to implementation of composed meta-views. For example, meta-views can be partially incompatible, and changes to specifications made according to one meta-view might affect—and even invalidate—other specifications according to other meta-views.

Meta-Tools

There are several meta-tools that implement the meta-views described in the previous section. We shall briefly examine four different meta-tool implementations, and shall relate them to their meta-views.

PROTÉGÉ

PROTÉGÉ [Musen, 1989a; Musen, 1989b] is a meta-tool that adopts a method-oriented view. PROTÉGÉ supports a particular problem-solving method for planning (skeletal-plan refinement), which also is the basis for the meta-view in PROTÉGÉ. Historically, PROTÉGÉ was abstracted from a domain-specific KA tool (OPAL) that acquires skeletal plans, or protocols, for cancer therapy. PROTÉGÉ incorporates an a priori design of target KA tools that is similar to the design of OPAL. The meta-view in PROTÉGÉ comprises concepts related to skeletal planning—for example, *planning entities* (which are processes that take place over finite periods of time), *task-level actions* (which are operations that control the planning entities and modify the plan during run time), and *input-data* specifications.

To build a KA tool with PROTÉGÉ, the developer must instantiate the skeletal-planning method supported by PROTÉGÉ for the domain in question. This instantiation involves describing planing entities, task-level actions, and input data in detail. PROTÉGÉ produces a target KA tool, which can be used by domain specialists to enter and edit skeletal plans, from the instantiated problem-solving method. In turn, the target KA tool produces the application system from the skeletal plans entered.

An important achievement of PROTÉGÉ is that it demonstrated how meta-tools can be used to instantiate KA tools from descriptions of problem-solving methods (i.e., PROTÉGÉ demonstrated the feasibility of the method-oriented view). Nevertheless, the principal drawback of PROTÉGÉ is inherited in its meta-view—the meta-tool is limited to one problem-solving method.

DOTS

DOTS [Eriksson, 1991] is a meta-tool that is based on the abstract-architecture view. Like PROTÉGÉ, DOTS is abstracted from a domain-oriented KA tool, but DOTS focuses on the architecture of the target tool, rather than on the problem-solving method of the application system. DOTS generates target KA tools from architectural specifications. Furthermore, DOTS assumes that the target tools conform to a particular architecture scheme (i.e., DOTS cannot be used to develop any type of software; it is tailored for development of graphical KA tools).

The meta-view in DOTS comprises (1) a variety of editors that can be custom-tailored to edit domain-specific structures, (2) a specification language for the internal representation (which represents what is entered in the editors internally) and other data structures for the target KA tool, (3) a set of update rules that can be configured to ensure consistency between the internal representations and the editors in the user interface, and (4) a set of transformation rules that is used to produce target code from the representation internal to the KA tool. To develop a KA tool with DOTS, the developer must analyze the domain and design a KA tool architecture for the domain, enter specifications for the domain-specific editors in DOTS, specify the internal representation for the target KA tool, declare the relationship between the editors and the internal representation in the form of update rules, and write transformation rules for code generation from the internal representation. DOTS produces a target KA tool from these architectural descriptions.

DOTS demonstrated how an abstract-architecture view can be implemented in a meta-tool. Unlike PROTÉGÉ, DOTS is not restricted to a particular problem-solving method or to any other domain class. DOTS, however, is restricted to a particular type of architecture for target KA tools.

SIS

Another meta-tool that implements an abstract-architecture view is SIS [Kawaguchi *et al.*, 1991]. SIS differs from DOTS in that it is designed for generating interview-based KA tools (i.e., KA tools that conduct a question-and-answer dialog with domain specialists to elicit domain information and knowledge), rather than graphical KA tools based on interactive editing for which DOTS is designed. The components of the architecture scheme supported by SIS, therefore, are different from those found in DOTS.

Spark

Researchers at Digital Equipment Corporation (DEC) have explored the organizational view as a basis for meta-tools. They have developed Spark, a meta-tool that implements the organizational view [Klinker *et al.*, 1991].

To implement a KA tool using Spark, the developer must identify the organizational type (e.g., manufacturing industry, service organization, or government), identify the role of the system in that organization using a diagram of typical organizations, and assemble a performance system using reusable program mechanisms from a library. Spark configures an appropriate KA tool from the description of program mechanisms and the information requirement for each of the relevant mechanism. The original Spark approach has been modified; the group at DEC is now considering mechanisms with a finer granularity.

Spark is part of a tool set that contains two other tools: Burn and FireFighter. Burn is the run-time system that controls the knowledge-acquisition session and invokes appropriate KA tools. FireFighter is a debugging tool that helps developers and domain specialists to debug and maintain application systems developed.

Programming Languages as Meta-Tools

General programming languages (e.g., C, Pascal, and ADA) also can be regarded as meta-views and their compilers can be seen as meta-tools, since they can be used to implement target KA tools. Programming languages, however, provide neither much support for tool implementation, nor any high-level constructs for tool specification (especially for interactive tools with graphical user interfaces). The use of programming languages can certainly provide flexibility in the tool design, but the implementation cost is often too high. Nevertheless, programming languages can play a role in implementation of tool functions that cannot be specified with an available meta-view.

Summary and Conclusions

Domain-specific development tools, including domain-oriented KA tools, are often reported to be more successful than are their general counterparts. Consequently, specialized development and KA tools are emerging. Since the development of such custom-tailored tools is relatively laborious given their restricted scope, researchers have experimented with meta-tools that support the design and implementation of domain-specific tools. Although it is preferable that meta-tools be domain-independent, their generality must be restricted if they are to be practicable and supportive. One such restriction is the class of target tools the meta-tool produces.

A meta-view is the specification strategy for target tools adopted by the meta-tool. The *method-oriented* view focuses on a problem-solving method that is applicable to many domains. The developer specifies domain-oriented target tools by instantiating a problem-solving method for the domain in question. The *abstract-architecture* view, on the other hand, focuses on the architecture of the target tool. In this approach, domain-oriented tools are specified through instantiation of architectural components (e.g., graphical editors, internal structures, and sets of transformation rules). The *organizational* view provides a model of generic organizations in which the role of the application system can be identified. Such roles are used as basis for generation of target tools.

The meta-views examined in this paper represents complementary approaches to specification of target tools. Since each meta-view has advantages and disadvantages, the choice of meta-view depends largely on the requirements on the target tool, development philosophy, and personal preferences. Ideally, meta-tools should support target-tool specification according to multiple paradigms.

With appropriate meta-tools, development of *application-specific* tools (rather than domain-specific) custom-tailored to particular development situations can be made feasible. Target tools can be changed during the course of the project to support different project stages in different ways. For example, target tools can serve as specification tools and then as maintenance tools, as the project evolves.

We are currently developing a meta-tool (PROTÉGÉ II) that will support a combination of meta-views [Puerta *et al.*, 1991]. PROTÉGÉ II will support two different development tasks simultaneously. One part of the emerging PROTÉGÉ II system will allow the developer to create basic performance systems by configuring tasks and problem-solving methods from a library of reusable components, the other of part PROTÉGÉ II is concerned with generation of domain-oriented KA tools (which are used for acquiring knowledge from domain specialists for the basic performance systems). For the KA-tool generation component, we are currently considering a combination of the abstract-architecture and ontological views. Since PROTÉGÉ II is also intended for configuration of tasks and problem-solving methods, the combined meta-view will incorporate ideas from the method-oriented view also.

Acknowledgments

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. We are grateful to Angel Puerta for comments on drafted versions of this paper and to Lyn Dupré for editorial assistance.

References

Eriksson, Henrik 1991. *Meta-Tool Support for Knowledge Acquisition*. PhD thesis 244, Linköping University.

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

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

Klinker, Georg; Bhola, Carlos; Dallemagne, Geoffroy; Marques, David; and McDermott, John 1991. Usable and reusable programming constructs. *Knowledge Acquisition* 3(2):117–135.

Marcus, Sandra, editor 1988. *Automating Knowledge Acquisition for Expert Systems*. Kluwer Academic Publishers, Norwell, Massachusetts.

Musen, Mark A.; Fagan, Lawrence M.; Combs, David M.; and Shortliffe, Edward H. 1987. Use of a domain model to drive an interactive knowledge-editing tool. *International Journal of Man–Machine Studies* 26(1):105–121.

Musen, Mark A. 1989a. *Automated Generation of Model-Based Knowledge-Acquisition Tools*. Morgan-Kaufmann, San Mateo, California.

Musen, Mark A. 1989b. An editor for the conceptual models of interactive knowledge-acquisition tools. *International Journal of Man–Machine Studies* 31(6):673–698.

Puerta, Angel R.; Egar, John W.; and Musen, Mark A. 1991. Automated generation of adaptable knowledge-acquisition tools with Mecano. Technical Report KSL-91-62, Knowledge Systems Laboratory, Stanford University, Stanford, CA.