Modern Architectures for Intelligent Systems: Reusable Ontologies and Problem-Solving Methods

Mark A. Musen, M.D., Ph.D.
Associate Professor of Medicine and Computer Science
Stanford Medical Informatics
Stanford University School of Medicine
Stanford, CA 94305-5479

When interest in intelligent systems for clinical medicine soared in the 1970s, workers in medical informatics became particularly attracted to rule-based systems. Although many successful rule-based applications were constructed, development and maintenance of large rule bases remained quite problematic. In the 1980s, an entire industry dedicated to the marketing of tools for creating rule-based systems rose and fell, as workers in medical informatics began to appreciate deeply why knowledge acquisition and maintenance for such systems are difficult problems. During this time period, investigators began to explore alternative programming abstractions that could be used to develop intelligent systems. The notions of "generic tasks" and of reusable problem-solving methods became extremely influential. By the 1990s, academic centers were experimenting with architectures for intelligent systems based on two classes of reusable components: (1) domain-independent problem-solving methods—standard algorithms for automating stereotypical tasks—and (2) domain ontologies that captured the essential concepts (and relationships among those concepts) in particular application areas. This paper will highlight how intelligent systems for diverse tasks can be efficiently automated using these kinds of building blocks. The creation of domain ontologies and problem-solving methods is the fundamental end product of basic research in medical informatics. Consequently, these concepts need more attention by our scientific community.

INTRODUCTION

Knowledge-based systems typically address reasoning tasks that are highly dependent on large amounts of domain information. The intelligent behavior of such systems requires the processing of great numbers of domain propositions organized into a knowledge base. How best to design electronic knowledge bases and to reason about them has been the subject of intense investigation within the knowledge-based–systems community for three decades. Workers have needed to develop new ways of managing complexity and of assuring maintainability over long system lifecycles. The need for systems to store so many domain facts and to reason about them to solve intricate domain tasks has demanded major advances in software engineering and software architecture.

Many workers in medical informatics think of knowledge-based systems in terms of the rule-based approaches that were popularized during the 1970s. The widespread availability of generic "shells" for constructing rule-based systems (such as OPS5 and CLIPS) and the advent of standards such as the Arden syntax[1] reinforces this notion. Although the majority of knowledge-based systems continue to be built using rule-based frameworks, there are well-known limitations to the scalability and maintainability of such systems. Simply put, the construction of large software systems by amassing unorganized collections of production rules is a problematic enterprise. The purpose of individual rules is often impossible to determine. Potential interactions among rules are often difficult to predict. Even the huge, commercial rule-based systems such as XCON, which provided some of the first convincing demonstrations of knowledge-based technology, became unmanageable without the imposition of considerable additional structure on the rule base.[2] Subsequent work to develop improved design methodologies for knowledge-based systems has emphasized techniques to manage complexity in the knowledge base and to clarify the way in which knowledge is used during problem solving.

GENERIC PROBLEM-SOLVING METHODS

By the middle of the 1980s, it was becoming apparent to many that there were recurring problem-solving strategies that were at the core of many knowledge-based systems. Clancey[3] proposed *heuristic classification* as a recurring inference pattern that could be identified in knowledge-based systems such as MYCIN and PUFF.[4] When performing heuristic classification,

1091—8280/98/$5.00

a problem solver took case data input into the system (e.g., a patient's white blood cell count) and abstracted those data into general descriptors (e.g., that the patient was a "compromised host"). Heuristics would link those abstractions to general candidate solutions (e.g., that bacteremia in compromised hosts often is caused by Gram-negative organisms), and additional knowledge would be used to refine general solutions into specific classifications (e.g., the patient most likely is infected with *E. coli*).

At around the same time, Chandrasekaran's group at Ohio State University identified several recurring problem-solving strategies in the medical knowledge-based systems that they were building. Chandrasekaran referred to these stereotypical problem-solving behaviors as *generic tasks*.[5] Meanwhile, John McDermott's group at Carnegie-Mellon University was noting a set of *problem-solving methods* that provided the control structure for a number of other knowledge-based systems built in non-medical areas.[6] All these investigators demonstrated that many intelligent systems had highly regular mechanisms for sequencing certain classes of inferences. These domain-independent problem-solving strategies provided standard ways of addressing certain kinds of application tasks. Even though the original developers of these knowledge-based systems might never have thought about these regularities explicitly, there were nevertheless a number of well-defined, generic strategies that were emerging from analysis of how diverse automated problem solvers addressed their associated application tasks. These generic strategies are now referred to as *problem-solving methods* by nearly all workers in the knowledge-based–systems community.

Problem-solving methods can provide a structure for building intelligent systems. When a designer can come to understand the domain knowledge needed to solve an application task in terms of a predefined problem-solving method, it becomes clear how each element of the domain knowledge might ultimately contribute to the problem-solving behavior of the system. When designing a heuristic classifier,[3] for example, the developer can identify readily whether a primitive inference is used to perform abstraction of case data into general features, heuristic matching of case descriptions to a possible solution, or refinement of a general solution into a more specific classification. The heuristic-classification model thus becomes a unifying framework by which to relate all the elements of domain knowledge that the developer might acquire. Using the problem-solving method as the basis for conceptual modeling limits the roles that domain knowledge can play in problem solving to those particular knowledge roles (e.g., feature abstraction, heuristic match) that are defined by the method. The *role-limiting* nature of problem-solving methods makes it clear what domain knowledge is needed to solve a task that can be automated with a given method (all the method's roles need to be filled for the method to work), and thus clarifies the purpose of each piece of elicited knowledge.[7]

When associated with a piece of program code that implements it, the problem-solving method becomes much more than an abstraction useful for conceptual modeling; the method becomes a building block in the design model that a programmer can use to implement a working system.[8] System builders can use the method conceptually to help them to model the domain knowledge that they need to acquire to build the decision-support system, and then can use the operational form of the problem-solving method to implement the decision aid. In this manner, the problem-solving method functions like an element from a mathematical subroutine library: It provides a reusable piece of software that facilitates implementation of the required computer program.

REUSABLE ONTOLOGIES

Chandrasekaran and his group made the claim that problem-solving methods (i.e., "generic tasks") imposed such strong assumptions about domain knowledge that is was impossible to think about domain knowledge independent from some problem-solving method[9]. They suggested that, because the number of distinctions that a modeler can make about the world is essentially infinite, one cannot begin to identify domain concepts relevant to a task until some problem-solving method has been selected. More importantly, because he believed that the meaning of the domain knowledge could become apparent only when that knowledge is applied to solve problems, Chandrasekaran argued that it was useless to begin to model application areas unless there was a commitment to a problem-solving paradigm up front.

Despite such concerns, workers in the 1980s came to accept that very general concepts about the application domain could be captured independently from the problem solver that ultimately might automate some task. More importantly, if the basic concepts in the application area could be represented within a separate, editable data structure, then the corresponding enumeration of basic application-area concepts might be reused to automate additional tasks. This point of view was fundamental to the highly influential KADS model

of knowledge engineering,[10] which encouraged developers to build models that made a clear distinction between basic domain concepts and the inferences and task-related procedures that might be applied to those concepts. In current parlance, these enumerations of domain concepts—and of relationships among the concepts—are referred to as domain *ontologies*.[11] An ontology provides a domain of discourse that is understandable by both developers and computers, and that can be used to build knowledge bases containing detailed descriptions of particular application areas.

A good example of a well-understood ontology is the categorization that Yahoo! provides users for searching the Internet. The Yahoo! ontology defines broad categories of entries on the World Wide Web. Users understand the ontology, and use it to locate the concepts that define their interests; the Yahoo! search engine also can process the ontology, and uses it to locate corresponding Web pages. The only relationships among the concepts in the Yahoo! ontology are taxonomic; there is no attempt to describe concepts in the ontology in terms of other relationships or attributes. Nevertheless, it is clear that the engineering of machine-processable ontologies has become a major business for companies promoting access to complex information sources. (The UMLS semantic network[12] provides a more modest example of how an ontology can assist information retrieval in clinical domains.)

The notion of reusable ontologies has become increasingly important to developers of intelligent systems. Just as modern database systems are driven by *conceptual schemas* that define the classes of entities about which the database stores specific data, modern knowledge-based systems incorporate as a central component of their knowledge bases a model of the classes of entities about which problem solving takes place. This ontology defines the relevant concepts and the attributes that concepts may have, but generally is silent regarding the specific values of the attributes that are assumed by particular instances of the concepts. For example, an ontology may indicate that there is a concept called *disease*, which has an attribute called *common name*; we would not expect the ontology to indicate that there is an instance of a disease called "streptococcal pharyngitis" when this disease has the common name of "strep throat." Of course, an ontology is not constructed without considerable forethought regarding what instances ultimately will need to be represented. Consideration of potential instances informs the developer's conceptualization of the classes that need to be included in an ontology; once the ontology has been defined, the instances are then represented in terms of the applicable classes and relationships.

Because ontologies are models, there is not a single correct way to define ontologies. There are many different perspectives that a modeler can take on a domain, and sometimes alternative perspectives may need to be captured simultaneously. An ontology represents a convenient way of characterizing a set of concepts and relationships in an application area. The merits of a particular ontology can be measured only in terms of how well that ontology supports development of the application programs for which it was designed, and of how easy it is for developers to reuse that ontology to build new applications.

Within the knowledge-based–systems community, there has been explosive interest in reusable ontologies in recent years. Developers are interested in the creation of libraries of ontologies that can capture the set of "core" concepts needed to model an application area, thus paving the way for reuse of previously developed ontologies within new systems.[13,14] Developers also see reference ontologies as a means for providing a canonical description of concepts that can allow integration of multiple information sources, particularly when individual information sources adopt idiosyncratic ways of referring to the same concept[15]. Within medical informatics, there is close association between the problems of creating controlled terminologies and those of building reference ontologies. Research projects such as GALEN[16], for example, are exploring how an ontology of clinical concepts can be used for a variety of purposes, including automated translation among vocabularies, structured data entry, and decision support.

KNOWLEDGE BASES

Because an ontology typically does not contain *instances* of concepts, we can view a knowledge base as an instantiation (or an extension) of an ontology. Thus, a knowledge base comprises "filled in" concept descriptions, enumerating the details of the particular application being built. In the EON system for protocol-based medical care,[17] for example, a general ontology defines the general structure of clinical protocols (the notions of drug therapy, laboratory tests, etc.); the particular knowledge bases on which EON operates, however, define specifications for particular protocols (i.e., individual protocols for AIDS, breast cancer, and so on).

Given a domain ontology, knowledge-acquisition systems such as Protégé[18] allow straightforward entry of the corresponding knowledge base. The

Protégé system permits developers to create a domain ontology using a simple editing system. Protégé then uses the domain ontology to create programmatically a user interface through which subject-matter experts can enter the detailed content knowledge required to populate a knowledge base. The tools generated by Protégé also can be used to browse and to update the knowledge base as necessary—provided that the overarching domain ontology remains constant. If the ontology should change, then it may be necessary to generate new knowledge-entry tools that capture the corresponding changes to the concepts and relationships in the ontology. (Of course, generating such tools automatically using Protégé is considerably more convenient than having to reprogram such tools by hand.)

PUTTING THE PIECES TOGETHER

In modern approaches, the building of intelligent systems is not to be construed as the encoding of production rules or the creation of specific knowledge representations. Rather, the process is viewed as the assembly of domain ontologies and (domain-independent) problem-solving methods. Given a task to automate, the challenge is to identify—or to construct—an appropriate problem-solving method, and to link that problem solver to an ontology that defines the relevant concepts in the application area. Thus, in our own laboratory, we have taken a general-purpose constraint-satisfaction problem-solving method known as propose-and-revise[19] and associated that method with a number of different ontologies to build a variety of systems. When the propose-and-revise method operates on an ontology of elevator parts, building codes, and engineering constraints, it automates the task of designing elevators for new buildings.[20] When propose-and-revise operates on an ontology that describes the molecular components of the *E. coli* ribosome and the kinds of constraints that experimental data place on the location of those molecular components in three-space, it automates the task of determining plausible conformations for the ribosome.[21] Similarly, we have developed appropriate ontologies to use propose-and-revise to address the tasks of ventilator management[22] and the planning of antiretroviral therapy for patients who have AIDS.[23] In building all these application programs, the propose-and-revise problem-solving method was completely reusable. The challenge in each case was in identifying how the generic data on which the propose-and-revise method operates can be related to the specific concepts in each domain ontology.

In general, as the knowledge-based–systems community has turned to building systems from reusable problem-solving methods and domain ontologies, the principal challenge has been to define the best way to glue the pieces together. In the approach that we have taken in the Protégé project[24], we have defined different types of *mappings* that provide the necessary relations. Thus, in systems built using Protégé, we create explicit objects that link particular data elements on which the problem-solving method operates to particular concepts in the domain ontology. Current work concentrates on refining an ontology of these kinds of mappings. This *mappings ontology* provides a structure for linking domain ontologies and problem-solving methods, guiding the process by which the two principal components are brought together within the same software system.

Workers on the Protégé project believe that maximal flexibility can be achieved by using declarative mappings to relate problem-solving methods to domain ontologies. The developers of the KADS methodology,[3] however, make no commitment to the manner in which ontologies and problem-solving methods should be brought together in an actual implementation. Other researchers, such as Fensel,[25] suggest that it may be more efficient to modify the problem-solving method directly by means of an *adapter* that can facilitate its interoperation with a domain ontology. Although there is considerable unanimity that domain ontologies and generic problem-solving methods provide the right kinds of abstractions for building intelligent component-based systems, there is less agreement on the optimal way for these individual components to communicate with one another. Unfortunately, in many knowledge-based systems that are constructed using discrete domain ontologies and problem-solving methods, the individual components are brought together in ad hoc ways that are dependent on the particular implementation environment.

DISCUSSION

For many years, workers in the field of conventional software engineering have anticipated a time when complex systems could be built rapidly by bringing together reusable software components. Although the general goals of software reuse have remained elusive,[26] the knowledge-based–systems community has achieved increasing success in reusing domain ontologies and problem-solving methods to craft new applications. The reuse of the propose-and-revise method to construct four different end-user systems provides an clear demonstration of how software components can be reapplied to construct programs for a variety of applications.[27]

Although the actual reuse of program code remains a difficult problem in conventional software engineering, there has been considerable interest recently in the reuse of *design patterns* that can help developers to structure their software solutions[28]. Design patterns in object-oriented programming provide exemplars to facilitate the conceptual modeling and implementation of software systems. The reusable domain ontologies and problem-solving methods being investigated by the knowledge-based systems community, however, allow much more direct reuse of previously tested solutions in the construction of new software. Domain ontologies and problem-solving methods provide not only a framework for conceptual analysis and design, but also actual program code that can be incorporated wholesale into evolving applications.

The use of distinct domain ontologies and problem-solving methods contrasts sharply with traditional object-oriented architectures, in which program code (in the form of "methods") is interleaved with the representation of the domain model (in the form of class and instance objects). Standard object systems facilitate specialization of program code on the basis of distinctions in the domain model (i.e., method polymorphism), and can clearly identify the data structures that are most closely related to the control structures that operate on those data. Such properties make traditional object-oriented systems rather malleable whenever programmers must adapt their software to evolving requirements. Object-oriented languages also make it relatively straightforward for developers to graft new functionality onto existing program code. At the same time, the tight linkage between the class hierarchies that constitute the domain model and the program code embodied in the objects' methods makes it difficult for an analyst to view either element independently. Thus, in a traditional C++ program, it is impossible to consider control-flow relationships separately from the data structures that comprise the object hierarchy. It also is impossible to view the data model without being distracted by associated program code.

The development of intelligent systems using separate domain ontologies and reusable problem-solving methods is particularly justified when the ontologies or the methods are likely to be reused to build new or derivative applications. The additional up-front engineering costs required to design the components for reuse then can be amortized over subsequent development efforts. This approach also is particularly attractive when developers can anticipate unusual requirements for software maintenance. The need for significant program revisions may result from domain models or domain-specific facts that are likely to evolve over time. In this situation, the incorporation of declarative domain ontologies can make domain-dependent information explicit, accessible, and easily editable, thus minimizing the difficulty of updating the system. A high degree of software maintenance may also be necessary when developers design new algorithms to achieve better performance; encapsulating an algorithm as a problem-solving method allows systems engineers to "plug in" new control strategies without having to alter either the domain ontology or the knowledge base.[29] (Of course, the mappings between to domain ontology and the new problem-solving method will need to be changed.)

At face value, the use of components such as ontologies and reusable problem-solving methods can make our software artifacts easier to build, easier to understand, and easier to maintain over time. These advantages occur because domain ontologies and problem-solving methods provide a means to view an intelligent system at a high level of abstraction—one that separates out the enumeration of domain concepts from the way in which those concepts might be used during problem-solving. Of course, the developers of rule-based systems in the 1970s contended that they had separated out "declarative knowledge" (in the form of rules) from "procedural knowledge" (in the form of inference engines that process those rules). Unfortunately, the inference engines of rule-based systems are special-purpose programs that operate on specific data structures (i.e., production rules). The rules, on the other hand, implicitly may encode considerable control-flow information.[30] In modern architectures, problem-solving methods are not procedures that operate on predefined data structures, but rather procedures that operate on *ontologies*.

This paper has emphasized the utility of domain ontologies and reusable problem solvers from a systems engineering perspective. We also must recognize, however, that these software components reflect conceptually the results of basic research in medical informatics. Our community continues to undergo considerable introspection regarding what the scientific contributions of medical informatics might be.[31] Development of information technology for clinical applications does not, at the surface, seem to involve the testing of hypotheses or the discovery of new theories. A unifying observation, however, is that research to understand the structure of medical knowledge ultimately requires the conceptual modeling of domain ontologies and the identification or invention of appropriate problem-solving methods. To build clinical in-

formation systems requires that we develop theories of how medical knowledge may be organized and processed. Such work requires that developers—either explicitly or implicitly—construct ontologies that reflect the way in which they construe medical knowledge. It also requires that the developers recognize problem-solving procedures that process the ontological knowledge in appropriate ways. Even if our domain ontologies and generic problem-solving methods could not be translated into useful and reusable software components, the elucidation of those ontologies and methods is an important scientific contribution. At the same time, because we can use such ontologies and methods as basic building blocks for constructing intelligent systems, there is now a direct mechanism to translate the conceptual results of our work into software artifacts that may have considerable clinical utility.

ACKNOWLEDGMENTS

This work has been supported by grant LM05708 from the National Library of Medicine, and by contracts supported by the Defense Advanced Research Projects Agency.

REFERENCES

1. Hripcsak G, Ludemann P, Pryor TA. Rationale for the Arden syntax. Comput Biomed Res. 1994; 27:291-324.
2. Bachant J. RIME: Preliminary work toward a knowledge-acquisition tool. In Marcus S. (ed), Automatic knowledge for acquisition for expert systems. Boston: Kluwer Academic Publishers, 1988; 201-24.
3. Clancey WJ. Heuristic classification. Artificial Intelligence, 1985; 27:289-350.
4. Buchanan BG, Shortliffe EH. Rule-based expert systems: The MYCIN experiments of the Stanford Heuristic Programming Project. Addison-Wesley, Reading, MA, 1984.
5. Chandrasekaran B, Johnson TR, Smith JW. Task-structure analysis for knowledge modeling. Communications of the ACM; 35(9):124-137.
6. Marcus S. (ed), Automatic knowledge for acquisition for expert systems. Boston: Kluwer Academic Publishers, 1988.
7. McDermott J. Preliminary steps toward a taxonomy of problem-solving methods. In Marcus S. (ed), Automatic knowledge for acquisition for expert systems. Boston: Kluwer Academic Publishers, 1988; p. 225-54.
8. Eriksson H, Shahar Y, Tu SW, Puerta AR, Musen MA. Task modeling with reusable problem-solving methods. Artificial Intelligence. 1995; 79:293–326.
9. Bylander T, Chandrasekaran B. Generic tasks for knowledge-based reasoning: The "right" level of abstraction for knowledge acquisition. In: Gains B, Boose J (eds). Knowledge acquisition for knowledge-based systems. London: Academic Press, 1988; p.65-77.
10. Schreiber A, Weilinga B, Breuker J (eds). KADS: A principled approach to knowledge-based system development. London: Academic Press, 1993.
11. Guarino N. Formal ontology, conceptual analysis and knowledge representation. International Journal of Human-Computer Studies, 1995;43:625-40.
12. McCray AT, Nelson SJ. The representation of meaning in the UMLS. Methods Inf Med, 1995;34(1/2):193-201.
13. Gruber TR. A translation approach to portable ontology specifications. Knowledge Acquisition, 1992; 5:199-220.
14. Guarino N. Understanding, building and using ontologies. International Journal Of Human-Computer Studies, 1997;46(2-3):293-310.
15. Arens Y, Knoblock CA, Shen WM. Query reformation for dynamic information integration. Journal of Intelligent Information Systems: Integrating Artificial Intelligence and Database Technologies, 1996;6(2-3):99-130.
16. Rector AL, Bechhofer SB, Goble CA, Horrocks I, Nowlan WA, Solomon WD. The GRAIL concept modelling language for medical terminology. Artif Intell Med, 1997;9:139-71.
17. Musen MA, Tu SW, Das AK, Shahar Y. EON: A component-based approach to automation of protocol-directed therapy. J Am Med Inform Assoc, 1996; 3:367–388.
18. Musen MA, Gennari JH, Eriksson H, Tu SW, Puerta AR. PROTÉGÉ-II: Computer support for development of intelligent systems from libraries of components. In: Proceedings of MEDINFO '95, Eighth World Conference on Medical Informatics. Vancouver, BC; 1995: 766-70.

19. Marcus S. SALT: A knowledge-acquisition tool for propose-and-revise systems. In: Marcus S. (ed), Automatic knowledge for acquisition for expert systems. Boston: Kluwer Academic Publishers, 1988;81-124.

20. Rothenfluh TE, Gennari JH, Eriksson H, Puerta AR, Tu SW, Musen MA. Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTÉGÉ-II solutions to Sisyphus-2. International Journal of Human-Computer Studies, 1996;44:303–332.

21. Gennari JH, Altman RB, Musen MA. Reuse with PROTÉGÉ-II: From elevators to ribosomes. Proceedings of SSR'95: ACM-SIGSOFT Symposium on Software Reusability. Seattle, WA, April, 1995, pp. 72–80.

22. Park JY, Musen MA. VM-in-Protégé: A study of software reuse. In: Proceedings of MEDINFO '98, 1998; in press.

23. Smith DS, Park JY, Musen MA. Therapy planning as constraint satisfaction: A computer-based antiretroviral therapy advisor for the management of HIV. In: Proceedings of the AMIA Fall Symposium, 1998, in press.

24. Gennari JH, Tu SW, Rothenfluh TE, Musen MA. Mapping domains to methods in support of reuse. International Journal of Human-Computer Studies. 1994;41:399-424.

25. Fensel D, Motta E. Structured development of protein solving methods. In: Gaines BR, Musen MA (eds). Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. Banff, Alberta, Canada, April 18-23, 1998.

26. Krueger CW. Software reuse. ACM Computing Surveys. 1992; 24:131-83.

27. Musen, M.A. Dimensions of knowledge sharing and reuse. Comput Biomed Res. **25**:435–467, 1992.

28. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. Pattern-oriented software architecture: A system of patterns. Chichester, UK: John Wiley & Sons, 1996.

29. Musen MA, Schreiber AT. Architectures for intelligent systems based on reusable components. Artif Intell Med, **7**:189–199, 1995.

30. Clancey WJ. The epistemology of a rule-based expert system: A framework for explanation. Artificial Intelligence. 1983;20:215-51.

31. van Bemmel JH. Medical informatics: Art or science? Methods Inf Med, 1996;35:157-72