Knowledge Engineering Tools at the Architecture Level

Thomas Gruber and Paul Cohen*
Experimental Knowledge Systems Laboratory
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

A *knowledge system architecture* is a level of description of knowledge systems that specializes general AI implementation techniques to suit a class of problem solving tasks. This paper presents three complementary views of the architecture level, and analyzes their implications for the design of knowledge engineering tools. The analysis is illustrated with an architecture for managing uncertainty by reasoning about actions, and with a hierarchy of knowledge engineering tools to support system development and knowledge acquisition at the architecture level.

I. Introduction

This paper is about tools for knowledge engineering at the *architecture level.* A knowledge system architecture specializes common AI problem-solving techniques to a particular class of tasks. An architecture provides descriptions of a particular kind of problem solving (e.g., *diagnosis* or *configuration*) at a conceptual level that is above the implementation, thus making clear which aspects of a class of problems are intrinsic to the problem and which are artifacts of the implementation. An architectures is a partial design of a knowledge system in which some decisions are made in advance to support particular task characteristics. For example, many medical diagnosis systems first interpret data bottom-up to find "triggered" disease hypotheses, then set top-down goals of acquiring evidence pro and con the triggered hypotheses. This "trigger/acquire evidence" cycle is an intrinsic part of any architecture for the class of medical diagnosis tasks, though it might be implemented in a wide variety of ways.

Architecture-level tools for knowledge engineers can improve the productivity of system development and knowledge acquisition because:

- By supporting the abstraction of representational and computational primitives at the architecture level, they permit the knowledge engineer and expert to cooperatively develop systems using a shared language of architecture constructs, rather than in terms of the underlying implementation.
- They can incorporate knowledge about the architecture to facilitate system development and knowledge acquisition (e.g., by enforcing constraints on the types and values of elements in the knowledge base).

The idea of an architecture level underlies recent work on knowledge systems.† Chandrasekaran and his colleagues have identified a number of "generic tasks" such as hierarchical diagnosis and routine design, and have developed task-specific representation languages and control strategies for them [Chandrasakeran, 1986, Bylander and Mittal, 1986, Brown and Chandrasakeran, 1985]. McDermott and colleagues have produced several knowledge systems using architectures that integrate knowledge acquisition tools with the problem solving methods [Kahn *et al.*, 1984, Eshelman and McDermott, 1986, Marcus, 1987, Kahn *et al.*, 1987]. Clancey has described in detail the heuristic classification method embodied in the HERACLES architecture [Clancey, 1986]. Newell [Newell, 1982] anticipated much of this work in his AAAI President's Address on the *knowledge level*, where he distinguished the knowledge of an intelligent agent, which is used to model its behavior, from the knowledge representation that describes how the knowledge is encoded in a symbol system.

This paper presents an analysis of the role of knowledge engineering tools at the architecture level. We describe three complementary views of what is meant by the architecture level, and illustrate them in the context of MU. MU is an architecture for systems that reason about the effects of actions to manage uncertainty. We show how the architecture-level analysis leads to a hierarchical organization of knowledge engineering tools to support software development and knowledge acquisition for MU systems. We conclude with some advantages of this approach to knowledge engineering.

II. Three views of the architecture level

Architectures can be viewed from three perspectives, and each suggests roles for architecture-level tools. First, the *functional* view presents an architecture as an application of general AI techniques to suit a particular style of problem solving. Described functionally, the blackboard architecture, for example, is well-suited to problems with noisy data and multiple sources of evidence. A knowledge system architecture specializes weak methods to solve a particu-

*This research is funded by National Science Foundation grant IST 8409623 and DARPA/RADC Contract F30602-85-C-0014.

†The architecture level was a major focus of the AAAI Workshop on High-level Tools in October, 1986. An earlier version of this paper was presented there.

lar class of tasks. Architectures have been developed for simple classification (e.g., decision trees), heuristic classification (e.g., HERACLES [Clancey, 1986]; CSRL [Bylander and Mittal, 1986]), constructing configurations (e.g., SALT [Marcus, 1987]; COAST [Bennett, 1986]), and routine design (e.g., DSPL [Brown and Chandrasakeran, 1985]; DOMINIC [Howe *et al.*, 1986]).

The second perspective is *structural*: an architecture is a partial design that includes specifications of knowledge representation formalisms, inference mechanisms, and control strategies. Many of the structural components, such as frame and rule systems, are provided by commercially available AI programming environments. Architectures, however, are not arbitrary combinations of these components, but artifacts designed by the knowledge engineer for particular tasks.

A third view of an architecture is that it defines a *virtual machine*. Just as Lisp provides primitives for symbol manipulation that the programmer can use without thinking about how they are realized in hardware, a knowledge system architecture presents representational primitives above the level of their implementation. The architecture provides a language that describes the behavior of a system in terms natural for the knowledge engineer and expert. For example, most medical diagnosis systems provide some kind of support for *triggering* – making particular hypotheses "active" when certain events occur, typically input data. To the expert, triggering might correspond to "bringing a diagnosis to mind." A programmer can produce the effect of triggering using implementation-level primitives (e.g., giving triggered diseases high certainty factors or agenda priorities). But terms such as triggering — not their implementation — are the medium of knowledge engineering. Such *task-level* terms promote explanation [Swartout, 1983] and knowledge acquisition [Gruber and Cohen, 1987]. Knowledge engineers, experts, and users can all understand triggering without thinking about how it is implemented. A virtual machine that executes triggering as a primitive is easier to program.

In summary, the functional view of an architecture emphasizes the behavior of programs that instantiate it. The structural view emphasizes knowledge representations, inference methods, and other components of the architecture. A virtual machine integrates these views: it is an abstract device designed to meet the functional needs of a class of problem solving tasks. The next section discusses how the interactions of these views result in an organization of knowledge engineering tools.

III. Tools for the MU Architecture

In this section we describe an architecture for systems that actively manage uncertainty, called MU [Cohen *et al.*, 1987b], with the aim of illustrating how the three views of architectures influence the design of knowledge engineering tools. MU grew out of experience with MUM (Managing Uncertainty in Medicine), a system for planning a series of diagnostic questions, tests, and treatments for diseases manifesting chest and abdominal pain [Cohen *et al.*, 1987a]. The primary aim of MUM is to decide how to act when data are insufficient for diagnosis and treatment. Like a physician, MUM reasons about tradeoffs between the costs of evidence, the marginal utility of potential data given what is already known, the effects of treatments and the evidence they provide, and so on. MU is an architecture for building systems like MUM that reason about uncertain situations in deciding how to act.

Viewed from a *functional* perspective, MU's task is *managing uncertainty* by taking appropriate actions. The task requires knowledge about the effects of actions on multiple goals, such as providing evidence for and against hypotheses, minimizing cost, and treating the condition. *Structurally,* MU has a large inference network of hypotheses, supporting evidence and intermediate conclusions, and actions that produce evidence and provide treatment; a working memory of developing hypotheses; inference mechanisms for propagating the effects of evidence in working memory; and support for strategies that choose among actions. Viewed as a *virtual machine,* MU supports knowledge engineering in terms that make sense for diagnostic tasks, such as *hypothesis* and *potential-evidence.* These terms are specialized for specific domains by terms such as *disease*, and further instantiated as specific diseases such as *angina.*

The interactions of these views of the MU architecture are apparent in the design of knowledge engineering tools. Figure 1 shows a hierarchy of tools that supports development of systems in MU. The foundation is a commercially-available AI programming environment that includes implementation primitives such as rules and frames, and basic AI programming techniques such as pattern-matching rule interpreters and message-passing. The bottom layer in Figure 1 is a structural description of the implementation of MU. It is not a design for an architecture, because no functional description has been given or is implied by this collection of implementation primitives, which could be instantiated to provide a wide range of behaviors.

The functional view of an architecture constrains how implementation-level primitives and techniques are specialized for a particular kind of problem-solving. The functional requirements of MU are that it should represent inferential relations among data, intermediate conclusions, and hypotheses. It should maintain measures of belief in all these objects, decide focus of attention (i.e., which objects to seek evidence for), and decide which evidence to seek. At the second level of Figure 1, the frames and slots of the first level are specialized as hypotheses and *inferential relations.* Inferential relations serve as pathways through the inference net. Rules are used to implement *combining functions* that specify how evidence supporting hypotheses is combined when propagated from subordinate nodes. Some properties of hypotheses and data-gathering actions — a subset of their slot values — are used as *control parameters*, which help determine focus of attention. The value propagation mechanism is implemented with the demons ("active values") and message passing capabilities of the frame system. In summary, the structure of the architecture is designed from implementation constructs to meet the functional requirements of a particular problem solving method, resulting in a virtual machine, or task-specific shell.

Tool Level	Objects in User's View	Software Support
Knowledge Acquisition Interface	*Domain-specific Terms* diseases, intermediate diagnoses, questions, clinical tests, triggering symptoms for diseases, confirming test results, criticality of diseases, relative costs of tests, treatments, efficacy of treatment	*(Meta-)Knowledge-based Utilities* language-specific editors and form-filling interfaces, inferential consistency analyzer, graphical display of the inference net
Virtual Machine (shell)	*Task-level Constructs* hypotheses, intermediate conclusions, data-gathering actions, inferential relations, combining functions, control parameters, control rules, preference rankings among actions	*Task-specific Reasoning Mechanisms* value propagation functions, predicates on the state of the inference net, rule-based planner, decision-making support
AI Toolbox (KEE)	*Implementation Primitives* frames and slots, rules, pattern matching language, Lisp objects and functions, windows and graphic objects	*AI Programming Techniques* knowledge base bookkeeping, rule interpreter, inheritance mechanisms, assumption maintenance, demon invocation and message passing, window system, network grapher

Figure 1: A hierarchy of knowledge engineering tools to support the MU architecture.

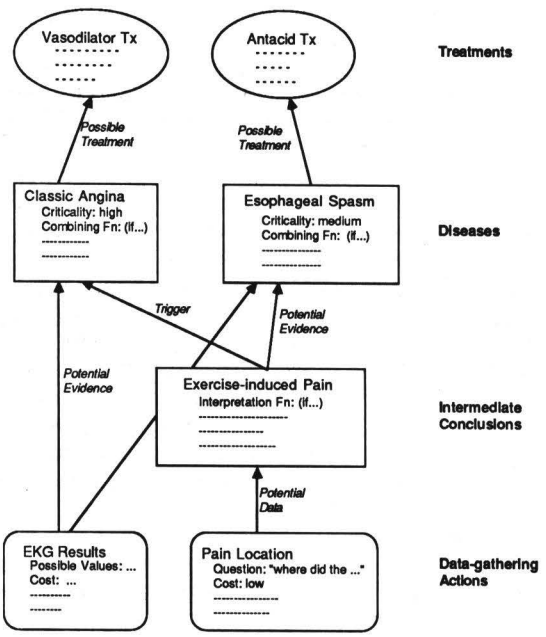

Figure 2: Fragment of the inference network for MUM

An architecture is designed not for a specific task like diagnosing chest pain, but for a class of tasks such as diagnostic reasoning. The knowledge engineer and expert instantiate architecture-level primitives for a particular application just as the architecture designer specializes implementation-level primitives. Figure 2 is a structural view of MUM – the chest pain specialist – engineered in the MU architecture. Hypotheses are instantiated as `diseases` such as `classic angina`; intermediate conclusions are instantiated as `clusters` such as `exercise-induced pain`; inferential relations are instantiated by specific links between evidence and conclusion, such as the `potential evidence` link between EKG `results` and `classic angina`.

Having instantiated architecture-level constructs such as hypotheses with domain-level terms such as diseases, the knowledge engineer can build a knowledge-acquisition interface to help elicit knowledge in the terms of the domain. Knowledge about the architecture-level terms is provided by the knowledge engineer in the shell, and is inherited by the domain objects used in an application. Knowledge acquisition utilities, on the top of the hierarchy, use meta-knowledge about objects in the knowledge base to help the user build a syntactically valid and semantically consistent knowledge base. Currently MU supports form-filling editors for all knowledge base objects, graphical interfaces for acquiring combining functions, and rudimentary consistency-checking abilities. Tools for interactively acquiring control knowledge are in progress.

IV. Conclusions

Architecture-level knowledge engineering tools have several advantages:

- One can capitalize on the vertical integration of implementation-level tools at the architecture level. For example, a general-purpose frame editor and network grapher provided at the implementation level (such as the KREME interface [Abrett and Burstein, 1987]) can be customized as a knowledge acquisition interface for editing architecture-level constructs such as hypotheses and their instantiations as diseases. This is possible because the architecture-level objects are specializations of implementation-level objects (i.e., frames), and consequently share their structure.
- Declaratively representing architecture-level constructs — the primitive objects of the virtual machine — encourages a consistent design shared by a team of programmers. For example, once the *trigger* relation has been designed, one need not worry about several members of a software project trying to achieve the same functionality with different implementations.

- Declarative architecture-level constructs also facilitate knowledge acquisition because meta-knowledge can be attached to objects to check for consistency, provide help, generate explanations, and so on. For example, a form-filling interface specialized for acquiring an instance of a disease can use a declarative description of the properties of diseases, such as the kinds of relations they have with data, to offer a menu of documented choices [Gruber and Cohen, 1987].
- Building a virtual machine at the architecture level and then a knowledge acquisition interface on top of the virtual machine defines the roles of the knowledge engineer and expert. The knowledge engineer *designs* an architecture by specializing general-purpose implementation-level tools to operationalize the constructs suited for the problem solving task, whereas the expert *instantiates* architecture-level constructs for the application domain. Virtual machine tools [sic] assist the knowledge engineer in customizing an architecture for a particular application, and knowledge acquisition tools help the expert build, refine, and debug the knowledge base.

V. Discussion

The hierarchy of tools discussed here reflects a power/generality tradeoff. Constructs at the implementation level are general (e.g., production systems can be configured for many kinds of problem solving) but from the standpoint of knowledge engineering they are weak. To say an object is a *disease hypothesis* is to imply much more knowledge about it than to say it is a *frame*, even though the implementation of the disease hypothesis may be no more than a frame. This added knowledge constrains the internal structure of the disease frame (e.g., values and types of slots, or the kinds of messages it can handle, etc.), constrains its relationships with other frames, and so on. Since these constraints facilitate knowledge engineering, architecture-level objects like disease frames are at the "power" end of the power/generality spectrum. Implementation-level objects, lacking constraints, are more general but correspondingly less powerful from the standpoint of knowledge engineering.

Thus, when one builds an expert system for a task, the utility of an architecture level analysis depends entirely on how much one *knows* about the task. The knowledge system architecture embodies knowledge about a class of problem solving tasks – it is a virtual machine for that class – and as such facilitates system development and knowledge acquisition for problem solvers of that class. The power/generality tradeoff tells us that we can ameliorate the knowledge acquisition bottleneck for restricted classes of tasks by designing architectures and building integrated "power tools" at the architecture level.

References

[Abrett and Burstein, 1987] Abrett, G. & Burstein, M. The KREME knowledge editing environment. *International Journal of Man-machine Studies*, in press.

[Bennett, 1986] Bennett, J. S. COAST: A task-specific tool for reasoning about configurations. Technical Report, Teknowledge Inc., Palo Alto, CA, 1986.

[Brown and Chandrasakeran, 1985] Brown, D. C., & Chandrasekaran, B. Expert systems for a class of mechanical design activity. In J. Gero (Ed.), *Knowledge Engineering in Computer-aided Design*, Amsterdam: North-Holland, 1985.

[Bylander and Mittal, 1986] Bylander, T. & Mittal, S. CSRL: A langauge for classificatory problem solving and uncertainty handling. *AI Magazine*, **7**(3), August, 1986, 66–77.

[Chandrasakeran, 1986] Chandrasakeran, B. Generic tasks in knowledge-based reasoning: high-level building blocks for expert system design. *IEEE Expert*, Fall, 1986, 23–30.

[Clancey, 1985] Clancey, W. J. Heuristic Classification. *Artificial Intelligence*, **27**, 1985, 289–350.

[Clancey, 1986] Clancey, W. J. From GUIDON to NEOMYCIN and HERACLES in twenty short lessons. *AI Magazine*, **7**(3), 1986, 40–60.

[Cohen *et al.*, 1987a] Cohen, P., Day, D., Delisio, J., Greenberg, M., Kjeldsen, R., Suthers, D., & Berman, P. Management of uncertainty in medicine. *Proceedings of the IEEE Conference on Computers and Communications*, Pheonix, Arizona, February, 1987, 501–506.

[Cohen *et al.*, 1987b] Cohen, P., Greenberg, M., & Delisio, J. MU: A development environment for prospective reasoning systems. *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, Washington, July, 1987, forthcoming.

[Eshelman and McDermott, 1986] Eshelman, L. & McDermott, J. MOLE: A knowledge acquisition tool that uses its head. *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, August, 1986, 950–955.

[Gruber and Cohen, 1987] Gruber, T. R. & Cohen, P. R. Principles of Design for Knowledge Acquisition, *Proceedings of the Third IEEE Artificial Intelligence Applications Conference*, Orlando, Florida, February 23–27, 1987.

[Howe *et al.*, 1986] Howe, A. E., Dixon, J. R., Cohen, P. R., Simmons, M. K. DOMINIC: A domain-independent program for mechanical engineering design. *International Journal for Artificial Intelligence in Engineering*, **1**(1), July, 1986, 23–29.

[Kahn *et al.*, 1987] Kahn, G. S., Breaux, E. H., Joseph, R. L., & DeKlerk, P. An intelligent mixed-initiative workbench for knowledge acquisition. *International Journal of Man-machine Studies*, in press.

[Kahn *et al.*, 1984] Kahn, G., Nowlan, S. & McDermott, J. A foundation for knowledge acquisition. *Proceedings of the IEEE Workshop on Principles of Knowledge-base Systems*, Denver, Colorado, December, 1984, 89–98.

[Marcus, 1987] Marcus, S. Taking backtracking with acquisition grain of SALT. *International Journal of Man-machine Studies*, in press.

[Newell, 1982] Newell, A. The knowledge level. *Artificial Intelligence*, **18**, 1982, 87–127.

[Swartout, 1983] Swartout, W. XPLAIN: A system for creating and explaining expert consulting systems. *Artificial Intelligence*, **21**(3), 1983, 285-325.