86.21

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

September 1986

1. Overview

This paper describes a organization for knowledge engineering tools based on the ***knowledge system architecture***, a level of description between weak methods and domain-specific applications. The architecture of a knowledge system is a partial design, in which some of the general design parameters such as knowledge representation formalism and reasoning mechanism are chosen in advance to support a particular class of problem solving tasks. An architecture supports constructs at a higher level than implementation primitives such as frames and rules, yet is not necessarily tied to a particular domain.

We present three ways of viewing an architecture: the functional view, the structural view, and the virtual machine view. We apply that analysis to MU, an architecture for designing knowledge systems that manage uncertainty by deciding how to act. We describe the role of knowledge engineering tools at the architecture level that can improve the productivity of system design and knowledge acquisition tasks. Distinctive features of architecture-level tools include

- They support an implementation of architecture-specific constructs, yet they make an explicit separation between implementation primitives and task-level terms, so that the tool user manipulates a language at the appropriate level for the task.

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

- They declaratively represent meta-knowledge about the evolving system, so that the design decisions are made explicit and can be used by other tools.

We show a hierarchy of knowledge engineering tools for supporting an architecture. For MU, the basic implementation level is supported by a commercially available AI programming environment. The architecture-level constructs defined by MU are provided by a runtime support shell, which operationalizes the architectural primitives while hiding the implementation. A knowledge acquisition interface presents the user with an environment for encoding knowledge in the terms of the architecture. Meta-knowledge about the terms is provided by the knowledge engineer while configuring the shell and is used by the knowledge acquisition interface to help the user build a syntactically valid and semantically consistent knowledge base.

2. The Architecture Level

A knowledge system architecture is a level of description of knowledge-based systems. It is a partial design of a knowledge system that applies a set of general AI techniques to solve a particular kind of problem. It explicitly supports a set of task-level constructs with which to model task-specific expertise. The architecture level terms used to represent task-level constructs provide a language for the knowledge engineer and expert independent of the implementation.

Postulating an architecture level is by no means novel, but it can stand some clarification. We will examine the notion of the knowledge system architecture from three views: as a task-oriented problem solving technique, as a partial design of a program, and as a virtual machine for representing problem solving knowledge.

1. **The functional view.**
   The architecture is a combination of AI techniques oriented toward a particular style of problem solving.
   There are architectures for simple classification (e.g., decision trees), heuristic classification (e.g., HERACLES, 1986), interpretation of noisy data (the blackboard model, Nii, 1986), and constructing a configuration (e.g., SALT, Marcus, McDermott, and Wang, 1985). An architecture may be more or less task-specific. The EMYCIN could be considered an architecture, or more accurately a pre-architecture, since although it was generalized from a medical diagnosis system it is fairly low-level (close to the implementation level of rules and certainty factors).[1] SALT is much more specialized, and can be more specific about the "roles of knowledge" it supports; it uses a task-specific algorithm that operates on knowledge of three well-defined types.

[1] Clancey claims that it is too low level – NEOMYCIN is closer to an architecture for medical diagnosis since it explicitly models the diagnosis process.

2. **The structural view.**
   The architecture is a partial design of a knowledge system.
   The architecture is designed to perform a particular task by choosing the appropriate combination of design parameters. Usually these include the *knowledge representation formalisms* (e.g., predicate calculus, production rules, semantic networks, frames in an inheritance hierarchy, measures of uncertainty); *reasoning mechanisms* (e.g., resolution theorem prover, forward and backward chaining rule interpreters, spreading activation, hierarchical matching, Bayes' rule); and *control strategies* (e.g., exhaustive search, chronological backtracking, agenda scheduling). A successful architecture is not an arbitrary collection of techniques, but rather a "good" combination, designed to do a particular task. A focus of current knowledge system research is to match tasks to existing architectures and to design new architectures when there is no good match to the tasks.

3. **The virtual machine view.**
   The architecture provides a language of task-level terms for representing knowledge.
   These terms describe the function of the system's design at a different level of abstraction, more natural for the knowledge engineer and expert, than the implementation level. While the architecture level may be *implemented* in computational primitives, in the sense of abstract data types, the constructs at the architecture level may be fruitfully distinguished from their implementation. This is the distinction that Allen Newell made more generally in his analysis of the *knowledge level* (Newell, 1982). He makes a clear distinction between the *knowledge* of an intelligent agent, which can be used to model its behavior, and the *representation*, which is how the knowledge is encoded in a symbol system.

   The architecture level can make a similar distinction. For example, most medical diagnosis systems provide some kind of support for *triggering* – making particular hypotheses "active" when certain events (typically input data) occur. To the expert, triggering might correspond to "bringing a diagnosis to mind"; for instance, a report of chest pain following exercise *triggers* the hypothesis of angina. Although the programmer may be able to produce the effect of triggering using implementation-level primitives (such as giving triggered diseases high certainty factors), the engineering detracts from the clarity of the term.[2] If an architecture provides task-level constructs as primitives, they become explicit terms for representing knowledge, which promotes explanation (Swartout, 1983) and knowledge acquisition (Gruber and Cohen, 1986). Knowledge engineers, experts, and users can all understand triggering without thinking about how it is implemented.[3] A virtual machine that executes "triggering" is easier to program.

---

[2] Also, this effectively bars the (non-programmer) expert from working with the knowledge base.

[3] MOLE (Eschelman and McDermott, 1986) has a 'domain model' of terms like "hypothesis" and "symptom" and weighted "paths", which it inherited from MORE (Kahn, Nowlan, and McDermott, 1984). One of the advances of of MOLE over MORE is to lessen the need for experts to manually set and adjust the numeric weights – clearly a implementation- rather than task-level representation.

3. The MU Architecture

MU is a knowledge system architecture that is being designed to support the use of domain knowledge to manage uncertainty. It 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, Day, Delisio, Greenberg, Kjeldsen, Suthers, and Berman, 1986). Diagnosis, or determining the causes for symptoms, is not the primary aim of MUM.[4] Instead, the system decides how to act when the data are insufficient to determine a disease. Like a physician, MUM must reason about the costs of gathering evidence (some tests for heart disease are dangerous and expensive), the marginal utility of the potential data given what is already known, the effect of treatments which may also provide evidence, and similar tradeoffs between the potential harm of being wrong about a dangerous disease and the effect of unnecessary treatment on the patient. Figure 1 shows a snapshot of MUM.

MU is a generalization of the techniques used in MUM.[5] It is a test bed for experimentation in managing uncertainty by reasoning about responses to uncertainty. As an architecture, MU can be examined from the three views of the architecture presented in the previous section.

3.1 MU's problem solving task

MU's task is *managing uncertainty* by taking appropriate action. Managing uncertainty includes heuristic classification (Clancey, 1985) as a subtask. The problem solver must assess the evidential support for possible hypotheses based on incomplete knowledge and insufficient data. MU's task is to decide what actions to take to respond to uncertain situations. Actions include gathering more evidence to reduce uncertainty about current hypotheses (e.g., by asking questions or running tests), prescribing treatment to reduce the effects of uncertainty (e.g., by treating for several possible diseases), or both (e.g., prescribing treatments which provide evidence for diagnosis). To decide among actions, the problem solver must reason about the marginal utility of possible actions. Actions must be proposed and the best actions selected.

The MU architecture is designed to perform this style of problem solving. It provides mechanisms to reason about partial evidential support for hypotheses, the effects of actions (especially on belief), and the marginal utility of actions. It is being explored as the architecture for knowledge systems in medicine, agriculture, and vehicle monitoring.

[4] Of course, given sufficient evidence, MUM can produce a diagnosis.

[5] much as EMYCIN generalizes MYCIN (van Melle, 1979). The difference is that EMYCIN is an implementation level tool (with primitive terms like rules, contexts, and certainty factors), whereas MU is an architecture (with terms like triggering, test, etc.).

3.2 MU as a partial design

The design of MU includes engineering decisions about knowledge representation formalisms, reasoning mechanisms, and control strategies.

Representation formalisms: MUM supports a symbolic inference net of frames connected by relations. Associated with inferential relations are levels of belief, represented symbolically, that are combined at nodes by local functions. Combining functions compute the level of belief of a node given the values of nodes on incoming inferential relation links. Combining functions are symbolic and are represented as sets of rules or as lookup tables. Knowledge about proposing actions is represented as frames containing predicates which match against states of the network and links to possible actions. Knowledge about selecting actions is represented as frames which associate actions with their effects (some of which are computed dynamically and describe their characteristics, such as cost.

Reasoning mechanisms: Values in the symbolic inference net are automatically propagated across inferential relations. For example, the value of the `evidential-support` slot in one node is propagated up the `potential-evidence` relation to another node. In that node, a combination function determines the effect of the incoming support, along with support from other potential evidence, and computes a support for the receiving node. This is the typical behavior of an inference net, except that in MU the values are symbolic, propagated by several types of relations, and each node has a local combining function. MU supports reasoning about actions with mechanisms for proposing actions given the current state of the network (e.g., what is known about diseases and actions), and a decision-making component for choosing from among proposed actions.

Control Strategies: MU is designed to incorporate domain-specific control knowledge. Thus its control strategy is not fixed, but parameterized by objects in the knowledge base. The top level loop is to evaluate the state of the network, match to frames which propose actions, evaluate the relative merits of proposed actions based on preferential knowledge (also stored in frames), and execute the actions.

3.3 Task-level primitives for MU

MU supports a network of typed objects connected by structural relations. Figure 2 shows the structure of a MU knowledge base. Although the knowledge engineer is free to define new types, the following have been sufficient to date. Hypotheses are the 'solution classes' of heuristic classification (Clancey, 1985). In MUM, diseases are hypotheses. Hypotheses are specializations of **clusters**, which are objects of belief supported by evidence. Clusters typically combine evidential support from several sources, including other clusters. In MUM, clusters are a natural representation for clinically significant groupings of evidence; they correspond to features of clinical situations to which the expert physician attends. **Data-descriptions**

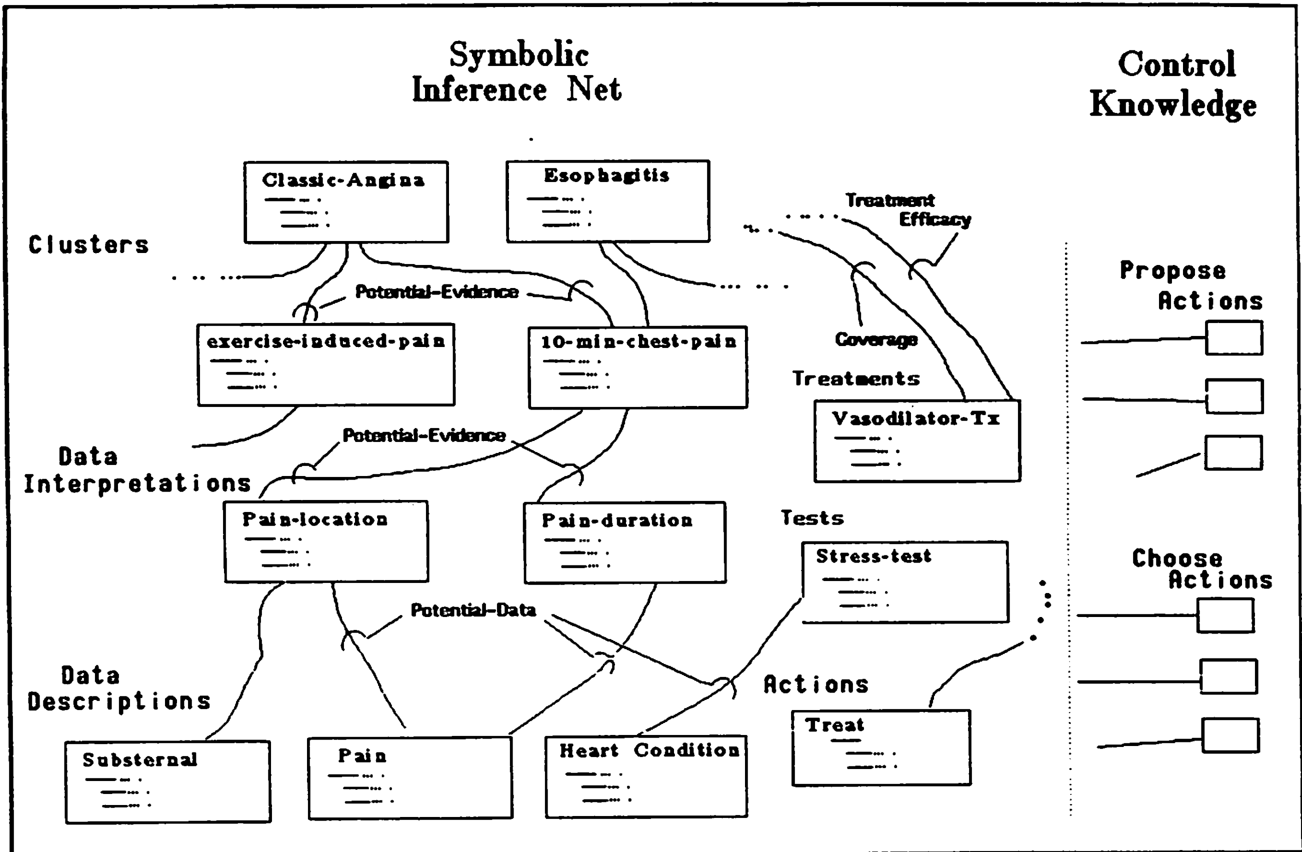

Figure 2: The structure of a MU knowledge base.

Objects in the symbolic inference net are connected by inferential relations that propagate values such as evidential support. The control knowledge is used to evaluate which actions to propose and which to choose, given the state of the net and characteristics of actions (tests and treatments are executed by actions).

represent knowledge about possible input: how to elicit it from the user, what answers are acceptable, etc. **Data-interpretations** are like clusters for input data; they are a means for expressing a categorical or definitional interpretation of one or more input data. The results of data-interpretations serve as evidence for clusters. For example, `angina-risk-factors` is a categorical assessment of the patient's age, gender, and family history. The specific values for the data (e.g., age) are summarized in a characterization of risk factors at a grain size appropriate to use as evidence for clusters (e.g., `classic-angina`). **Actions** are events that the system can prescribe. Tests are a kind of action. So are therapies. Actions have effects, such as causing new data to come in, which potentially change the state of the system. Primitives for reasoning about actions are called **control parameters** (Wesley, 1983). Those used to propose actions are tests on the state of the inference net, such as the current support for clusters, and the characteristics of those clusters, such as criticality. Control parameters for choosing from proposed actions are characteristics of actions, like cost and reliability, and the effect of actions on goals, like the efficacy of treating a disease or the diagnosticity of the data returned by a test.

Objects in MU are associated by relations that represent automatic inferences. The **potential evidence** relation associates clusters with the nodes that might contribute to their evidential support. When the belief in an inferior cluster or data-interpretation node changes, the value is propagated up the potential-evidence links to the clusters they support (or detract). The **triggering** relation also relates data-interpretations and clusters to other clusters, but instead of inferring evidential support, they simply cause the clusters to be "triggered", which is a control parameter for proposing actions. The **coverage** relation relates treatment actions to the clusters that they "cover". **Potential-data** relations associate data-descriptions with their data-interpretations. Relations can be composed from these primitives, such as the potential evidence produced by the data that could result from performing an action.

4. An organization of architecture support tools

The MU architecture has led naturally to a hierarchy of software support, starting with the familiar implementation-level tools[6], upon which is built a architectural-level virtual machine ("shell"), and finally a knowledge acquisition interface for building systems using the architectural constructs. Figure 3 shows the relationship among the knowledge engineering tools supporting MU.

Implications of architecture level analysis for the design of high level knowledge engineering tools.

Tools should support the connection between the implementation level and the architecture level. This is the role of the so-called "shell" – to implement a virtual machine that executes the task-level primitives using the partial design to achieve high performance on the problem solving

[6]currently we use KEE, a product and trademark of IntelliCorp

A Hierarchy of Architecture Support Tools		
Tool	**Objects in User's View**	**Implementation tasks**
AI toolbox (KEE)	*Implementation Primitives*	*AI Programming Techniques*
	Rules	Rule interpreters
	Frames and slots	KB Bookkeeping
		Inheritance Mechanisms
	Lisp objects and functions	Demon and message passing support
Virtual Machine (shell)	*Architectural primitives*	
	Inferential Relations, Combining Functions	Custom Match/propagation functions
	Control Parameters	Predicates on state of inference net
	Control Rules	Action-proposing function
	Preferences	Decision-making support
Knowledge Acquisition Interface	*Application-specific Terms*	*(meta-)Knowledge-based utilities*
	Diseases, tests, treatments, questions, intermediate diagnoses, criticality of diseases, costs of tests, Efficacy of treatment	Knowledge-base form-filling editors, consistency checker, graphical display for objects and relations

Figure 3: A hierarchy of knowledge engineering tools to support the MU architecture.

task. Each task-level construct defined for the architecture should have an implementation in the shell. For example, an inferential relation in MU is an architectural primitive, yet to implement it requires defining the relationship between four slots in two class frames, a syntax and matcher for the combining functions (in Lisp), and demons (data-directed Lisp functions) for propagating values over the relation. The engineering decisions that go into implementing architecture primitives are done once, and the knowledge engineer is then free to concentrate on more knowledge-level tasks.

Tools should should support a knowledge acquisition interface that makes a clear distinction between the implementation and architecture levels. The interface should present a "user illusion" (Kay, 1984) that the machine is operating at the architecture level, hiding the implementation. This is an application of the principles of "design for acquisition" (Gruber and Cohen, 1986) – knowledge acquisition is better understood if the architecture provides representational terms that are natural for the expert. For example, in EMYCIN, control knowledge is typically implemented by ordering the production rules. If the expert wanted to convey that some questions should be asked together, or that some should only be asked if others return a particular answer, then the programmer would have to find some ordering if the rules (possibly in conjunction with using markers and adjusting certainty factors) that produce the desired behavior. In MU, control knowledge can be expressed explicitly with control parameters and rules for proposing

and choosing actions.

The architecture level constructs should be represented as declarative objects. The primitive objects and operations that an architecture defines can be implemented using opaque Lisp code. Indeed, from the expert's view it makes no difference how they are implemented. From an knowledge engineering standpoint, however, giving task-level terms a declarative representation allows the knowledge engineer to attach meta-knowledge to them. That meta-knowledge can be used to produce explanations of the runtime performance of the application (Swartout, 1983), but also to make the design of the architecture itself explicit and explainable (Neches, Swartout, and Moore 1984; Mostow and Swartout, 1986). A common kind of meta-knowledge is information about the type and value restrictions of terms that the expert will instantiate when building a knowledge base. In MU, the knowledge engineer defines methods for data acquisition (simple prompted type-in, various flavors of menus, plotting points on a graph, filling a form) and the methods for presenting help information for each term.[7] This "extra" description is more than commenting the code; it can be used to enforce consistency at the implementation level.

Meta-knowledge about architectural terms – those primitives of the virtual machine – can be used by the knowledge acquisition interface to constrain the user's input and explain to the user the purpose and use of the primitives. Randy Davis' thesis (Davis, 1976) made this point using MYCIN's rules as the primitives supported by TEIRESIAS. Sandra Marcus' work on SALT (Marcus, McDermott, and Wang, 1985; Marcus, Caplain, McDermott, and Stout, 1986) has demonstrated the principle at a different point in the power/generality tradeoff. By supporting less general but more task-specific primitives such as "propose a constraint", the knowledge acquisition tool is able to assist the user with the content, not just the form, of entries in the knowledge base. For MU, providing strong typing and other meta-knowledge about architectural terms has made it possible to use standard data entry technology – essentially form-filling – to acquire knowledge from experts.[8] Figure 4 shows an invocation of a form-like rule editor on a combining function for computing evidential support. It can insure that every operation used to build a combining function is valid and consistent. For example, the editor restricts the values of terms in the left hand side of a combining function to those defined by meta-knowledge description of the evidential relation. If the user wants to create a new term, the editor can then invoke the proper "form" for instantiating the appropriate node – in this case it is a cluster. This technique works the same way as the NLMenus (Tenant, 198?) approach to natural language interface. The problem of too much expressive power in the language[9] is engineered away by restricting the user to a valid choices. The constraints are one application of meta-knowledge about task-level terms.

[7]Some task-level terms are implemented as a slot in a class of frames, and others require several data structures and procedures, such as the inferential relations described above.

[8]Of course this is the "easy" kind of knowledge acquisition. The more difficult tasks of designing the architecture and defining the appropriate task-level primitives is resisting automation.

[9]Users of natural language systems often overestimate the grammatical coverage of the system.

```
(Output) The CLASSIC-MI Unit in DOCTOR Knowledge Base
Unit: CLASSIC-MI in knowledge base DOCTOR
Created by FREED on 29-Jul-1986 19:22:25
Modified by GRUBER on 28-Aug-1986 20:29:37
  Member Of: DISEASE

  Classic Myocardial Infarction

Own slot: COMBINING-FUNCTION from CLASSIC-MI
  Inheritance: OVERRIDE.VALUES
  Comment: "Specifies how evidence is combined to affect the level of
upport for this frame"
  Values: (IF (CONFIRMED EATING-PAIN-RELIEVED-BY-POSITION) THEN
           STRONGLY-DETRACTED)
          (IF (CONFIRMED RECLINING-CHEST-PAIN) THEN
           STRONGLY-DETRACTED)
          (IF
           (AND (CONFIRMED ANGINA-RF-WITH-AGE)
            (CONFIRMED TEN-MIN-PAIN-RELIEVED-BY-NITRO)) THEN
           STRONGLY-DETRACTED)
          (IF (CONFIRMED PLEURISY) THEN DISCONFIRMED)
          (IF (CONFIRMED ANGINA-RF-WITH-AGE) THEN
           STRONGLY-DETRACTED)
          (IF
           (AND
            (CONFIRMED ISCHEMIC-CHANGES-
            (CONFIRMED TEN-MIN-PAIN-RELIE
           STRONGLY-DETRACTED)
          (IF
           (AND
            (CONFIRMED
             PAIN-OR-PRESSURE-GTR-10-MIN-
            (STRONGLY-SUPPORTED DAF-EKG-FOR-CLASSIC-MI)) THEN
           DETRACTED)
          (IF
           (AND
            (CONFIRMED
             PAIN-OR-PRESSURE-GTR-10-MIN-NOT-RELIEVED-BY-NITRO)
            (STRONGLY-SUPPORTED DAF-EKG-FOR-CLASSIC-MI)) THEN
           STRONGLY-SUPPORTED)
          (IF (CONFIRMED DAF-EKG-FOR-CLASSIC-MI) THEN CONFIRMED)
          (IF (CONFIRMED CONFIRM-MI-CLUSTER) THEN CONFIRMED)

Own slot: CRITICALITY from CLASSIC-MI
  Inheritance: OVERRIDE.VALUES
  Comment: "A measure of the prognosis"
  Values: HIGH

Own slot: EVIDENCE-FOR from CLUSTER
  Inheritance: OVERRIDE.VALUES
  ValueClass: INFERENCE-FRAME
  Comment: "If this frame plays a role of potential evidence for any oth

Choose a BELIEF-STATES
  Unknown
  CONFIRMED
  DETRACTED
  DISCONFIRMED
  STRONGLY-DETRACTED
× STRONGLY-SUPPORTED
  SUPPORTED
```

```
Rule Editor for Combining Function of Classic-MI
IF <potential evidence> IS <belief state> and...
THEN CLASSIC-MI IS <belief state>

IF EATING-PAIN-RELIEVED-BY-POSITION IS CONFIRMED and...
THEN CLASSIC-MI IS STRONGLY-DETRACTED

IF RECLINING-CHEST-PAIN IS CONFIRMED and...
THEN CLASSIC-MI IS STRONGLY-DETRACTED

IF ANGINA-RF-WITH-AGE IS CONFIRMED
AND TEN-MIN-PAIN-RELIEVED-BY-NITRO IS CONFIRMED
THEN CLASSIC-MI IS STRONGLY-DETRACTED

IF PLEURISY IS CONFIRMED and...
THEN CLASSIC-MI IS DISCONFIRMED

IF ANGINA-RF-WITH-AGE IS CONFIRMED and...
THEN CLASSIC-MI IS STRONGLY-DETRACTED

IF ISCHEMIC-CHANGES-WITH-PAIN-OR-PRESSURE IS CONFIRMED
AND TEN-MIN-PAIN-RELIEVED-BY-NITRO IS CONFIRMED
THEN CLASSIC-MI IS STRONGLY-DETRACTED

IF PAIN-OR-PRESSURE-GTR-10-MIN-RELIEVED-BY-NITRO IS CONFIRMED
AND DAF-EKG-FOR-CLASSIC-MI IS STRONGLY-SUPPORTED
THEN CLASSIC-MI IS DETRACTED

IF PAIN-OR-PRESSURE-GTR-10-MIN-NOT-RELIEVED-BY-NITRO IS CONFIRMED
AND DAF-EKG-FOR-CLASSIC-MI IS STRONGLY-SUPPORTED
THEN CLASSIC-MI IS STRONGLY-SUPPORTED

IF DAF-EKG-FOR-CLASSIC-MI IS CONFIRMED and...
THEN CLASSIC-MI IS CONFIRMED

IF CONFIRM-MI-CLUSTER IS CONFIRMED and...
THEN CLASSIC-MI IS CONFIRMED
```

Figure 4: A knowledge-based rule editor instantiated on a combining function for a disease cluster.

The window on the left shows the implementation-level (KEE) view of the same data structure. The terms in the rule editor window can be selected and edited. The domain of legal values for each term is represented declaratively and used by the editor. In the example, the user has selected a right-hand side term of a rule, which for this kind of combining function must be a member of the class called `belief states`. The editor can insure that the user enters a valid belief state by using a menu interface.

5. Discussion

The idea of an architecture, while not new, has implications for the design and organization of knowledge acquisition tools, and thus for knowledge engineering practice. Once an architecture is in place, knowledge acquisition will be facilitated, for all the reasons discussed above. Two issues must be addressed, however, before the architecture level supports a knowledge engineering methodology. The first is *design* and it has two aspects. First, the architecture level can be viewed, as we noted in Section 2, as a partial design *at the implementation level* for a knowledge-based system. But clearly, knowledge engineers should not be required to map architecture level constructs (such as trigger) to their implementation before deciding on an architecture for an application. Instead they should select a partial design for an application based on an architecture-level description of the representations and inference methods required. The architecture-level constrains the knowledge engineer more than the implementation level, so the criteria for a "good fit" between an application and a shell are more stringent than, say, between an application and a production system. Thus, one aspect of design is that the knowledge engineer is more constrained in selecting a shell, and this is positive, except in those cases that *designers* fail to give the most general description of their architectures. This is the second aspect of design. For example, if we say the MU shell supports triggers but fail to say that triggers are an instance of a class of evidential relationship supported by MU, then the prospective knowledge engineer will be misled. What we require is a kind of axiom of cooperativeness where tool designers fulfill the expectation to describe their tools at the most general level they support.

The second issue for architecture-level tools is *maintenance* by iterative debugging and knowledge acquisition. We believe that the architecture level implies better explanations of system performance, and so a better debugging environment. TEIRESIAS both explained and debugged MYCIN by unwinding a goal stack, which unavoidably involved the expert at the implementation level. Our long-term goal is to explain and debug entirely at the architecture level. For example, if a conclusion is accepted when it should not be, we want the expert to be able to say that evidence should have "ruled out" the conclusion without worrying about how "ruling out" is implemented.

References

Clancey, W. From GUIDON to NEOMYCIN and HERACLES in twenty short lessons. *AI Magazine*, 7(3), August 1986, 40-60. See the references in this report for details.

Clancey, W. J. Heuristic Classification. *Artificial Intelligence*, **27**, 1985, 289-350.

Cohen, P., Day, D., Delisio, J., Greenberg, M., Kjeldsen, R., Suthers, D., & Berman, P. Management of uncertainty in medicine, 1986. Department of Computer and Information Science, University of Massachusetts, COINS Technical Report 86-12.

Davis, R. Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases. Doctoral dissertation, Computer Science Department, Stanford University, 1976. Reprinted in R. Davis & D. B. Lenat (Eds.), *Knowledge-Based Systems in Artificial Intelligence*, New York: McGraw-Hill, 1982.

Eshelman, L. & McDermott, J. MOLE: A knowledge acquisition tool that uses its head. *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, PA, August 1986.

Gruber, T. and Cohen, P. Design for acquisition: Designing knowledge systems to facilitate knowledge acquisition, 1986. Department of Computer and Information Science, University of Massachusetts, COINS Technical Report 86-21. To be presented at the Knowledge Acquisition Workshop, Banff, Canada, November 1986.

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.

Kay, A. Computer Software. *Scientific American*, **251**(3), September 1984, 52-59.

Marcus, S., McDermott, J., & Wang, T. Knowledge acquisition for constructive systems. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985, 637-639.

Marcus, S., Caplain, G., McDermott, J., & Stout, J. C. Making SALT Generic. Department of Computer Science, Carnegie-Mellon University, 1986.

Mostow, J. and Swartout, W. Towards explicit integration of knowledge in expert systems: an analysis of MYCIN's therapy selection algorithm.

Neches, R., Swartout, W. R., & Moore, J. Explainable (and maintainable) expert systems. *Proceedings of the IEEE Workshop on Principles of Knowledge-base Systems*, Denver, Colorado, December 1984, 173-184.

Newell, A. The knowledge level. *Artificial Intelligence*, **18**(?)), 1982, 87-127.

Nii, H. P. Blackboard Systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, **7**(2), Summer 1986, 38-53.

van Melle, W. A domain independent production rule system for consultation programs. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence,* August 1979, 923-925.

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

Tenant, H. NLMenus. ACL Proceedings, 1983?

Wesley, L. P. Reasoning about control: The investigation of an evidential approach. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* August 1983, 203-206.