A Problem Space Approach to Expert System Specification

Gregg R. Yost and Allen Newell

School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

One view of expert system development separates the endeavor into two parts. First, a domain expert, with the aid of a knowledge engineer, articulates a procedure for performing the desired task in some external form. Next, the knowledge engineer operationalizes the external description within some computer language. This paper examines the nature of the processes that operationalize natural task descriptions. We exhibit a language based on a computational model of problem spaces for which these processes are quite simple. We describe the processes in detail, and discuss which aspects of our computational model determine the simplicity of these processes.[1]

1. Introduction

Viewed abstractly and somewhat simplistically (Figure 1, top), one fundamental paradigm of expert system development starts with a domain expert who *articulates* the means of performing the task in some language T. A knowledge engineer then comprehends the task knowledge expressed in T, resulting in a conceptualization of the knowledge in terms of the task domain TD. Next, the knowledge engineer maps the task knowledge from the terms of TD to the terms of some computational domain CM (called a *computational model).* Finally, the knowledge engineer composes a set of statements that express the computational conceptualization of the task in a computer language L. Together, the comprehension, domain mapping, and composition are referred to as *operationalization* of the task knowledge. This description does not imply that all task knowledge is articulated before operationalization begins. In practice, these phases are highly interleaved and incremental. The processes described apply to individual knowledge fragments, not to the body of task knowledge as a whole.

T must be a language that both the domain expert and the knowledge engineer are familiar with, and that permits clear and concise description of the task knowledge. Thus, T is usually a natural language. In the remainder of this paper we assume that T is English. Operationalization remains a task for humans, rather than computers, because natural language comprehension is routine for humans but is much too difficult to perform automatically. Further, operationalization remains a task for knowledge engineers, rather than domain experts, because the latter rarely are skilled in the use of computer languages. Thus, this paper assumes a human knowledge engineer and an appropriate level of language skills in T and in L (it also assumes that the description of task knowledge in T does not pose its own difficulties by being a confusing or obscure text). The remainder of this paper focuses on the third component of operationalization: the conceptual mapping from the task domain to the computation domain.

By separating the notion of a language from the domains it describes, we see that improving the state of the art in expert system development is not simply an issue of making language improvements. We may modify a language so that it describes its domains more perspicuously, but the fundamental conceptual mismatch between the task domain and the computation domain remains. This conceptual mismatch accounts for most of the difficulty of operationalization. If the processes that perform the mapping are complex and open ended, then operationalization will be a difficult intellectually-creative task. If these processes are simple and routine, then design of expert systems will be routine. Different computational models could require quite different processes, and thus could present quite different degrees of difficulty.

This paper illustrates this thesis by describing PSCM *{Problem Space Computational Model),* a computational model with a small set of clearly-describable operationalization processes (Figure 1, bottom). PSCM is a computational model of problem spaces based on the Soar architecture (Laird, Newell & Rosenbloom, 1987, Laird, 1986). TAQL *{Task Acquisition Language)* is the language (L) that is based on PSCM; it has a compiler that

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499, and monitored by the Air Force Avionics Laboratory. The research was also supported in part by Digital Equipment Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, Digital Equipment Corporation, or the U.S. Government.

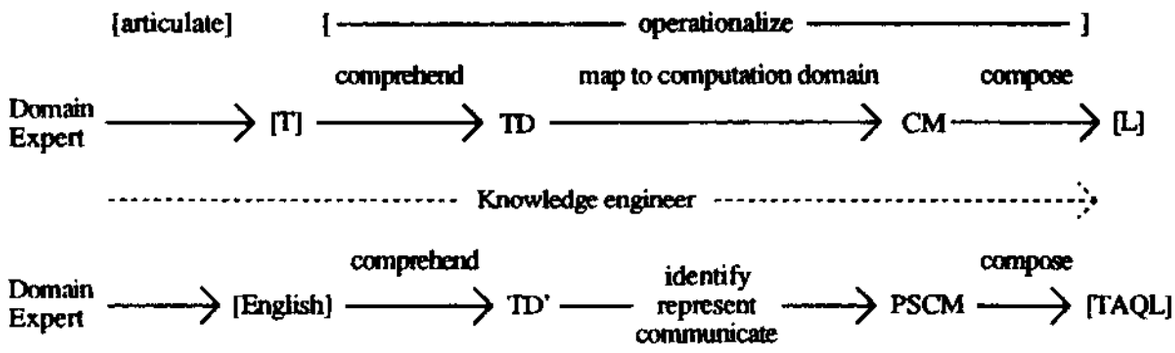

Figure 1: A fundamental paradigm of expert system development (simplified).

converts systems described in TAQL into Soar, hence into running expert systems. Only three types of processes are required to operationalize an English task knowledge description into TAQL: *identification, representation* and *communication,* each of limited character and complexity (given adequate language skills in English and in TAQL).

Section 2 describes PSCM. Section 3 describes the operationalization processes. Section 4 describes TAQL and gives a brief example of operationalization. Section 5 discusses the role of the computational model in determining the operationalization processes.

2. The Computational Model

A computational model is a set of entities, some of which perform operations on other domain entities. Thus a computational model has two kinds of domain entities: *structural* entities, and *functional* entities. Structural entities are the basic objects in the domain. Functional entities perform operations on structural entities. Some entities may be both structural and functional. For example, in the computational model underlying Lisp, certain lists may be treated as either programs or data.

PSCM is a computational model of problem spaces abstracted from the Soar problem solving architecture. In PSCM, as in Soar, all tasks are represented as finding a desired state in a problem space.

Table 1 lists the entities that comprise PSCM. The first row of the table lists the structural entities: tasks, problem spaces, states, and operators. Tasks are particular problems to be solved. Problem spaces are organizing structures that group related knowledge. States consist of the data objects relevant to the task. Operators manipulate states with their associated data objects. The rest of the table lists the functional PSCM entities. They are grouped by the structural entity with which they are most closely associated. In the rest of this paper, these structural and functional entities are collectively referred to as *PSCM components.*

Each PSCM component has a set of aspects that must be defined for the component to be meaningful. For example, an operator proposal component has three such aspects: which problem space the component belongs to; the conditions under which the operator should be proposed; and the operator object to be proposed, expressed in terms of objects in the current problem-solving context.

A task is represented as a collection of interacting problem spaces, each of which performs some portion of the task. Problem spaces interact in a variety of ways. For example, one problem space may implement an operator invoked in another problem space. During problem solving, problem spaces are situated within a goal hierarchy. Whenever a new goal is created, problem solving proceeds in that goal as follows:

Select a problem space and initialize it
Select an initial state for the problem space
While no goal test is satisfied:
  Propose operators to apply to the current state
  Select a single operator designated as better than all others by the available operator selection components
  Apply the operator to the current state, producing a new current state

Subgoals are generated whenever problem solving in the current problem space cannot proceed until another problem space has performed some subtask on its behalf. For example, when the available operator selection components do not uniquely determine which operator should be applied next, PSCM creates a subgoal to choose one of the candidate operators.

Problem solving in PSCM proceeds in general without knowledge of a global control structure for the task. Rather, PSCM assembles a solution dynamically through the application of a sequence of localized problem-solving components. Some sets of PSCM components may lead PSCM to exhibit the behavior produced by a well-known problem solving method. Other sets of components may exhibit no easily-characterizable global behavior. However, PSCM admits problem-solving methods that influence the overall behavior of an entire set of components. Such method-based behavior is easily produced, but is not required.

While PSCM and Soar are both computational models of problem spaces, PSCM describes tasks at a higher level of abstraction than Soar. Soar expresses problem space computations in terms of concepts such as productions,

Tasks	Problem Spaces	States	Operators
Goal testing Methods	Problem space proposal Problem space initialization	State initialization Elaboration Evaluation	Operator proposal Operator selection Operator implementation Operator failure Evaluation

Table 1: Components of the PSCM computational model

working memory, preferences, and impasses. PSCM abstracts away from these architectural mechanisms and describes problem spaces in their own terms.[2]

3. The Operationalization Processes

The processes that operationalize English descriptions of domain knowledge into a given computational model are determined by the computational model. As displayed in the lower half of Figure 1, the operationalization of English task knowledge into TAQL is produced by a knowledge engineer who comprehends the knowledge description, maps the comprehended task concepts to components of PSCM, and composes a set of TAQL language statements expressing those PSCM components. For PSCM, the mapping between domain and computational model performs three functions: *identify* a PSCM component; *represent* a data object; and *communicate* some information from one PSCM component to another.

In general, while the operationalization processes themselves are determined by the computational model, their instantiation and application to a given knowledge description is strongly determined by the forms of expression used in that description. For PSCM, we can make an even stronger statement: *for descriptions of real-world tasks expressed naturally and in their own terms, the operationalization processes yield a set of PSCM components that directly model the forms of expression used in the description.* In other words, to a large extent, the processes involve not design and creative reformulation, but comprehension and re-expression of English knowledge descriptions in the terms of PSCM. This is particularly true for the identification and representation processes. The remainder of this paper explicates this claim.

To begin, we describe the three processes in more detail.[3] Let E denote an English description of the task knowledge for a particular task. The first function to be performed is to identify PSCM components in E. All types of PSCM components are identified at this stage, including organizational components such as problem spaces; data-object components that make up the problem solving state; and problem-solving methods, which determine the behavior of an entire set of PSCM components. The identification proceeds by labeling paragraphs, sentences, or phrases in E with the PSCM components that will encode the knowledge in those parts of E. In essence, it involves segmentation of the text in E.

The labels are assigned based on comprehension of the functional roles of parts of E. For example, a description of how to perform some subtask would be labeled with the name of an operator to perform that subtask, and would be classified as an operator-implementation component. Components created for related subtasks are grouped into problem spaces. A method is identified when E describes behaviors that match the behaviors known to be produced by the method. A structure that is the target of some action described in E is identified as a data object (part of a state). The identification of a data object may be further refined by classifying it as an instance of an abstract data type: such an identification is made when E describes manipulations of the identified data object that match the computational operations defined on the abstract data type.

After identification, the next function to be performed in the operationalization of E is to *represent* data objects. The identification process yields a conceptualization of the task knowledge in terms of abstract problem spaces. At this stage, most of the procedural structure of the final PSCM solution has been determined. Tasks have been assigned to operators, related subtasks have been grouped into problem spaces, and the relationships among problem spaces have been determined at an abstract level. The interactions among operators within a space are also known at an abstract level. However, the interactions among problem spaces and operators cannot be completely determined until data representations have been selected. Immediately after identification has completed, objects are still in terms of the task domain, except for the occasional appearance of abstract data type terms.

Data representatioas require raw materials out of which they can be constructed. The choice of raw materials depends to some extent on what is appropriate for the computational model. For example, representations built from machine-level units such as bytes (e.g., records and arrays) have proved appropriate for the computational models underlying conventional programming languages such as C and Pascal. For PSCM, the representations are in terms of attribute/value structures, which have

Soar also provides a learning mechanism (chunking). PSCM does not.

[3] An example of these processes in action appears in Section 4.

historically proved useful in computational models for expert systems (e.g., OPS5 (Forgy, 1981)) and are used in Soar.

The representations of data objects described in E are developed in the same way as the PSCM components were assembled. That is, the attribute/value structures are not so much *designed* as they are *identified* from the structure of their descriptions in E. Thus, if E mentions a backplane with nine slots, it might be represented as an object of class *backplane* with a *slots* attribute whose value is 9. These structures can be hierarchical. For example, if E mentions individual backplane slots and their widths, the backplane object may be given a *slot* attribute, the value of which is an object of class *slot* having a *width* attribute. In the cases where the abstract data type of an object has been identified, representation is even easier: it is assumed that the knowledge engineer is skilled in the expression of common abstract data types in the terms of PSCM; thus, creativity is not required.

Once the representation process has been completed, the operationalization of E in PSCM is almost finished. Most of the interactions among PSCM components are known by the time the identification process completes, and the components need only be restated in terms of the chosen attribute/value representations to become operational in PSCM. However, since the components were identified at an abstract level (before data representations were known), some of these components may now need to be modified or refined.

This fine-tuning of interactions is the province of the *communicate* process. Communication comes in two forms: inter-space communication, and inter-operator communication. Both forms of communication are driven by the need to make available the information operators must have to apply correctly and in the proper order.

The abstract problem space descriptions classify the connections among problem spaces. For example, they may state that the problem space in a subgoal implements a specific operator in the superspace. However, they do not indicate in any detail what information in the current goal needs to be made available in the subgoal, or what information produced by the subgoal needs to be returned to the supergoal when the subgoal exits. The *communicate* process fills in the details of this inter-space communication.

Data objects are copied *to* a subgoal to make them readily available to the problem space operating in the subgoal, or to any of its subspaces. This is particularly important for data objects that are modified by operators in the subspace. Data objects are copied *from* a subgoal to the superspace either to make a result available to operators in the superspace, or to preserve the value of a data object for a future invocation of the same problem space or one of its subspaces.

Inter-operator communication must be refined in two situations:

1. When an operator needs data in a form other than the form created by the operator that produces the data*
2. When an operator needs data that was available at some point during prior computation, but that would not otherwise be preserved in the current state.

The first situation can be resolved by either modifying the operators involved so that they represent the data in the same way, or by introducing a new operator or elaboration component that translates between the two forms. The second situation can be resolved by modifying operators that had access to the required information in the past so that they make this information part of their result states, thus preserving it for future use.

4. The TAQL Specification

The operationalization processes described in Section 3 map task knowledge from the terms of the task domain into the terms of PSCM. The final requirement in the operationalization is to express these computational model structures in a formal, compilable language: namely, TAQL.[4] The stages of the complete operationalization process are displayed graphically in the left-hand column of Figure 2 (we will describe the rest of that figure below).

TAQL directly reflects the structure of PSCM. Thus, a TAQL specification consists of a set of *TAQL constructs,* called TCs, each of which describes some aspect of a PSCM component. A Common Lisp program compiles TCs into Soar productions. When loaded into Soar along with a small set of runtime support productions, these productions implement the task described by the TCs. This compilation is fully automated and very efficient: it does not take noticeably longer to load a file of TCs than it does to load the productions generated by those TCs.

Each TC is a list consisting of the TC type and a name for the TC instance, followed by a list of keyword arguments. Each keyword specifies some aspect of the related PSCM component. An operator-proposal TC appears at the bottom of Figure 2. In terms of PSCM, the aspects that must be defined for an operator-proposal component are the problem space it applies in, the operator object to be proposed, and the conditions under which the operator should be proposed. These aspects are specified directly in the *propose-operator* TC as the values of the *:space, :op,* and *.when* keywords, respectively. Data is represented in TAQL using attribute/value structures of the form produced by the representation process during operationalizati on.

We now provide a detailed example of the operationalization of a small piece of English task

[4]See the TAQL User Manual (Yost, 1988) for a more detailed description of TAQL than is given here.

[5]The current version of TAQL makes no attempt at graceful syntactic forms, as the emphasis so far has been on the operationalization processes.

description. The domain is computer configuration, as performed by the R_1/XCON expert system (McDermott, 1982). RI was coded in OPS5. Several years ago, the unibus-configuration subtask of RI was recoded in Soar (Rosenbloom, Laird, McDermott, Newell & Orciuch, 1985). RI-Soar is an expert system of about 340 rules. Since its creation, it has served as a testbed for a number of efforts within the Soar project.

We have produced an English description of the unibus configuration task, and have realized this task in TAQL by applying the operationalization processes to that description. Figure 2 illustrates how the operationalization processes apply to a small piece of the description. The two English sentences at the top of the figure express when specific instances of an action (backplane cabling) should be performed. This is exactly the kind of information a PSCM operator-proposal component expresses. Thus, the identification process yields two operator-proposal components, one for each of the two cable lengths. Only the component for cables of length ten is shown in detail in the figure. Next, the representation process applies to the conditions in the abstract component, and also to the operator object that is to be proposed. A straightforward mapping from the English description of the abstract component yields the attribute/value representations shown.[6]

The condition that determines whether or not the backplane has been filled with modules is naturally expressed as a test for the presence of a *^modules-configured* attribute on the state. However, the operationalization of the modules-into-backplane operator, which fills the backplane[7], does not generate this information. Thus, the communication process must build a link between the modules-into-backplane and cable-bp operators. It does so by modifying modules-into-backplane to return the required modules-configured attribute, in addition to any other actions the operator already performs.

Before leaving this example, we say a few words about how R1-TAQL compares to R 1-Soar. For the comparisons given here, we use an updated version of RI-Soar that reflects the task-oriented conceptual structure of unibus configuration more closely than the original RI-Soar did.[8]

Both RI-TAQL and RI-Soar use the same seven problem spaces. R1-TAQL has 153 TCs, and RI-Soar has 337 hand-coded Soar productions. The 153 TCs compile into 352 Soar productions. A more useful measure of size is the number of *words* in each description, where a word is defined to be the smallest unit that has meaning to the TAQL compiler or to the Soar production compiler. Words include attribute names, variables, and parentheses, among other things. The English description of RI has 756 words; R1-TAQL has 5774 words, and RI-Soar has 21752 words. Thus the number of words in RI-TAQL is 26% of the number of words in RI-Soar, a significant reduction.

[6]Part of the representation of the configuration structure is determined by other parts of the RI English task description (not shown), and is simply reused here.

[7]This operator is described in a part of the RI description not shown here.

[8]This was the joint work of Amy Unruh and Gregg Yost.

5. The Role of the Computational Model

We have described PSCM and TAQL, a computational model and associated language that require only simple processes of operationalization. Existing practice takes the creation of expert systems to be a difficult task, although the development of knowledge acquisition tools and expert system shells has simplified the task for some classes of systems at the expense of generality in the tool (Clancey, 1983, McDermott, 1988). Much of this difficulty resides in operationalization, although articulation of domain knowledge (which is outside the scope of this paper) clearly contributes as well.

The desirable course would be to describe the operationalization processes for existing expert system specification languages, and compare them with the processes for TAQL. However, this course is not presently feasible, because the operationalization processes for other languages cannot yet be specified. All that is known is the overall complexity of the language in practice. For example, Common Lisp, a standard, highly effective, general purpose system-building language, still requires substantial effort when used to build medium-sized expert systems. But to give the operationalization processes for coding expert systems in Common Lisp would be to describe how to do program synthesis of very substantial and complex programs — well beyond current understanding. That operationalization processes can be described when they are simple (as they are for TAQL) does not imply that they can be described when they are more complex.

However, some things can be said. For most specification systems, the basic computational model is some variant of procedural semantics: data types with associated sets of operations, on top of which is provided a set of procedural control constructs, built out of the notions of execution, sequence, and conditionally. Production systems, object-oriented programming, conventional programming, and a number of other schemes are all variations on this theme. All such specification systems require specifying such things as programs, methods, strategies, reasoning schemes, executives, etc. For simple applications, this may be easy, but as the complexity of the application grows this becomes a genuine program design task. The operationalization processes for using these languages must include some way to synthesize the required methods, executive organizations and so forth. Let us call this operationalization process *method-design.*

Method-design is not required for operationalizing into TAQL. This is surely a major factor in the simplicity of its operationalization processes. Some of the reasons for this are presented in Section 2: the mutually supporting problem space structure of PSCM provides the

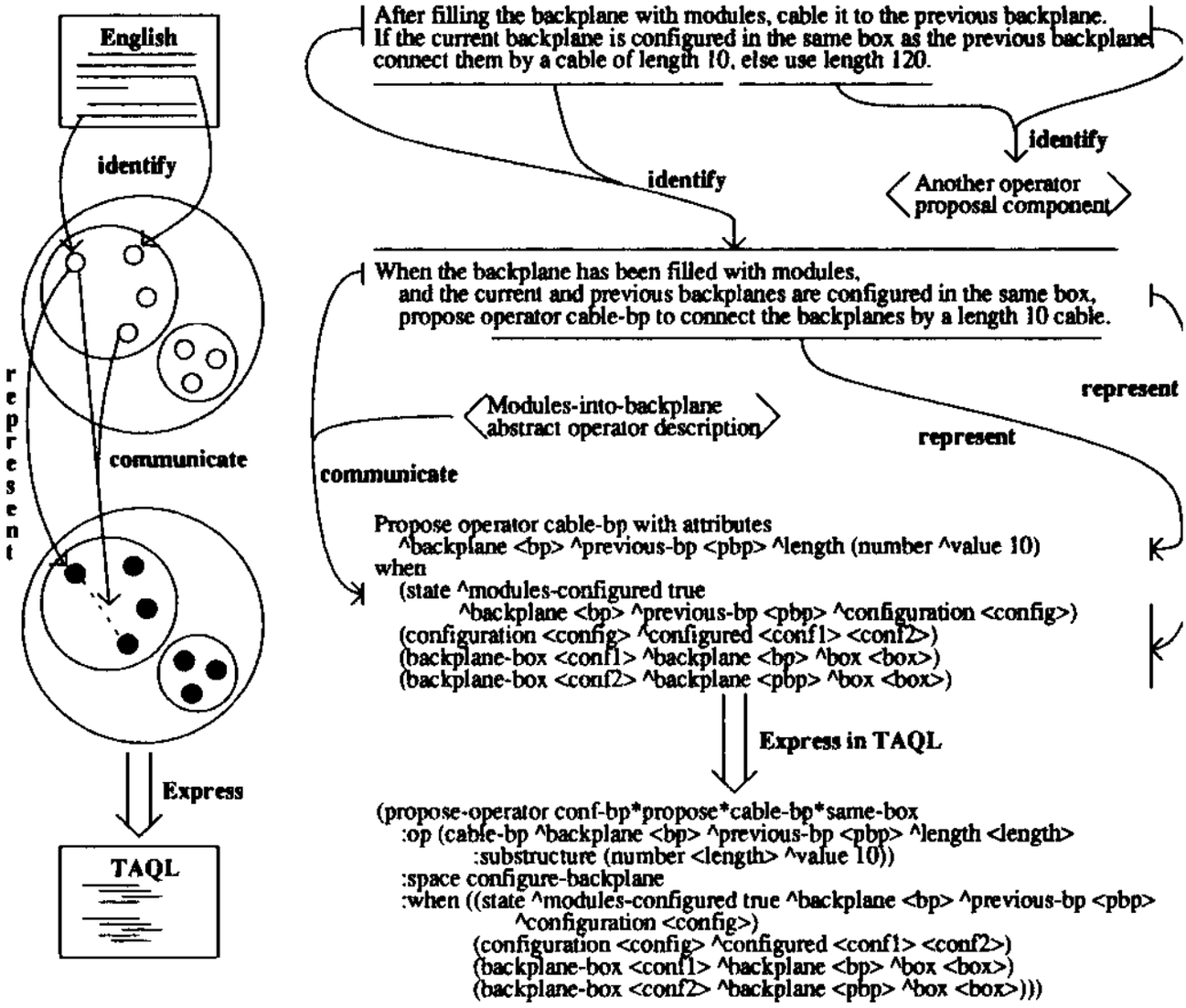

Figure 2: The operationalization of part of R1 in TAQL

organization into which local control knowledge can be placed without having to design any methods or higher-level organizations. In this respect, PSCM differs from RIME (Bachant, 1988), a programming methodology based on problem spaces and in some ways similar to PSCM. In RIME, each problem space is required to specify a single method that will determine the course of problem solving in that space. In PSCM (as in Soar), method-based behavior is an emergent phenomena of localized problem-solving decisions.

Of course, methods can be quite useful. Once coded, they can be reused in similar tasks by simply modifying a few bits of domain knowledge. Methods determine the behavior of an entire set of problem solving components, and they can provide very concise specifications in appropriate situations. However, if the desired task behavior does not very closely match the method's behavior, it can be quite difficult to force the task to fit the method. The key point here is that PSCM provides the flexibility to either use methods or not, depending on which is most appropriate in a given situation. TAQL does provide a set of methods, which can be used when appropriate. However, only one situation occurs in R1-TAQL where the use of a method (limited depth-first search) is more appropriate than a customized set of components. An important result of our work is that the nature of the PSCM operationalization processes facilitates the selection of appropriate sets of these customized components.

6. Conclusions

This paper has exhibited a computational model and associated formal language for which the processes of operationalizing naturally-expressed expert system task knowledge are quite simple, in particular avoiding method-design while having a quite general scope. Our explicit focus on the processes that transform from the task domain to the computation domain is a departure from much of the expert system specification literature. Many efforts are

involved with the invention of new formal languages and the description of the processes, sometimes quite complex, by which those languages are reduced to some previously-known operational computing system (for example, GIST (London & Feather, 1982) and KEE (Filman, 1988)). While such work is both interesting and useful, it is often left to the reader's intuition to see why it might be easier to use the new language rather than some existing language. Our study of the processes that construct TAQL specifications from an informal English description of task knowledge is an attempt to articulate these intuitions for a particular computational model (PSCM).

Our future work will proceed along two paths. First, we plan to build a tool that helps a knowledge engineer carry out the PSCM operationalization processes. The knowledge engineer will bring natural English descriptions of task knowledge to the tool. The tool will help select and apply appropriate iastances of the identification, representation, and communication processes, ultimately producing a TAQL implementation of the task. We will evaluate the effectiveness of our tool with respect to existing tools over a wide range of tasks.

We do not believe it will be possible in the near future to fully automate operationalization in a general-purpose expert system development tool, and we will not attempt this. The language skills required are well beyond the state of the art. Many research effort resolve this problem by limiting the generality of the tool. We take a different approach. Our tool will leave the language skills with a human knowledge engineer, who can perform them routinely. We focus instead on the aspect of operationalization people find most difficult: mapping knowledge from task domain terms to computational terms. The PSCM operationalization processes that have been the focus of this paper seem sufficiently limited that a computer can provide strong guidance in their application.

The second path we intend to explore is more theoretical. Now that we have some understanding of the operationalization processes for TAQL, we want to discover what aspects of the underlying computational model determine the simplicity of these processes. As discussed in Section 5, that PSCM does not require method-design is one such aspect. However, there are surely others we have not yet identified. We also want to explore whether the processes we have identified apply to tasks substantially larger than the unibus configuration task, and, if not, we want to explore what higher levels of organization might be required for large tasks.

Acknowledgments

We wish to thank Amy Unruh, for rewriting much of RI-Soar to correspond to our English description of the task; Erik Altmann, Thad Polk, and Milind Tambe, three TAQL users who have provided much valuable feedback; Paul Rosenbloom, for his comments on earlier drafts of this paper; and John McDermott for helping us understand the RI task, for his insights into the nature of the expert systems development, and for his continued assistance.

References

Bachant, J. (1988). RIME: Preliminary work toward a knowledge-acquisition tool. In Marcus, S. (Ed.), *Automating Knowledge Acquisition for Expert Systems.* Boston, MA: Kluwer Academic Publishers.

Clancey, W. (1983). The advantages of abstract control knowledge in expert system design. *Proceedings of the Third National Conference on Artificial Intelligence.* Washington, D.C..

Filman, R. E. (1988). Reasoning with words and truth maintenance in a knowledge-based programming environment. *Communications of the ACM,* 57(4), 382-401.

Forgy, C. L. (July 1981). *OPS5 user's manual* (Tech. Rep. CMU-CS-81-135). Carnegie Mellon Univereity, Computer Science Department.

Laird, J. E. (1986). *Soar User's Manual: Version 4.0.* Intelligent Systems Laboratory, Palo Alto Research Center, Xerox Corporation.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33(1),* 1-64.

London, P. and Feather, M. (1982). Implementing specification freedoms. *Science of Computer Programming,* 2(2), 91-131.

McDermott, J. (1982). RI: A rule-based configurer of computer systems. *Artificial Intelligence, 1*9(l), 39-88.

McDermott, J. (1988). Preliminary steps toward a taxonomy of problem-solving methods. In Marcus, S. (Ed.), *Automating Knowledge Acquisition for Expert Systems.* Boston, MA: Kluwer Academic Publishers.

Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., and Orciuch, E. (1985). RI-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 7(5), 561-569.

Yost, G. R. (1988). TAQL 2.0: Soar Task Acquisition Language User Manual. Computer Science Department, Carnegie Mellon University, October, 1988.