CEST: AN EXPERT SYSTEM FUNCTION LIBRARY AND WORKBENCH FOR *UNIX*® SYSTEM/C LANGUAGE

William B. Frakes and Christopher J. Fox

William B. Frakes *is supervisor of the Intelligent Systems Research Group in the Quality Theory and Technology Department at AT&T Bell Laboratories in Holmdel, New Jersey, and* ***Christopher J. Fox*** *is a member of technical staff in the Intelligent Systems Research Group. They work on knowledge-based tools for C language, incorporating nontextual information into knowledge bases, and software engineering. Mr. Fox joined the company in 1985 and has a B.A. and M.A. in philosophy from Michigan State University, and an M.S. in computer science and Ph.D. in information science from Syracuse University. Mr. Frakes joined the company in 1982 and has a B.A. in liberal studies from the University of Louisville (Kentucky), an M.S. in library science from the University of Illinois,* (continued on page 106)

Integrating expert system components into production software can be difficult, because environments for developing expert systems typically are not compatible with traditional software-engineering technology. To deal with this problem, we are developing CEST, a C-language expert system toolset. It is a library of inference engines—implemented as C functions that can be called from C programs—and a workbench of knowledge-engineering support tools. CEST allows easy integration of expert system components into C-based software systems, and provides knowledge-engineering support tools analogous to traditional software-engineering support tools. The first tool written for CEST is AVIEN, a backward-chaining attribute-value inference engine. It has been widely distributed within AT&T, and has been used to build both stand-alone expert systems and C-based hybrid systems. In particular, the Quality Assurance Center at AT&T Bell Laboratories is using AVIEN in software tools being developed for quality and reliability analysis.

Background

In the past few years, expert system technology has spawned enormous interest. Attempts are currently underway to apply this technology to a wide range of problems. But because of efficiency and portability constraints and the dissimilar software-development environments for procedural and declarative languages, expert system techniques can be difficult to incorporate into production software.

In this paper, we describe work on CEST (C expert system tools), an expert-system function library and workbench for the UNIX® system and C-language environment. (Panel 1 defines acronyms used in

this paper.) The CEST project grew out of a need to add expert system capabilities, such as intelligent user interfaces and problem solving strategies, to software tools under development at AT&T Bell Laboratories. Like most AT&T software tools, these new tools are C based, must run under UNIX system, and must be of production quality (that is, robust, portable, maintainable, etc.). The CEST project, therefore, directly addresses the problems of providing expert system tools and techniques for production-quality software development. Because we present a viable technology for integrating expert components into production software, our work has importance beyond the UNIX system and C environment.

Panel 1. Acronyms in This Paper

AAAI	American Association for Artificial Intelligence
ACE	Automated Cable Expert
ACM	Association for Computing Machinery
AI	artificial intelligence
ASQC	American Society for Quality Control
AVIEN	attribute-value inference engine
CEST	C expert system tools
Cobol	common business-oriented language
ESD	expert system development
Fortran	formula translator
GUESS	general-purpose expert system shell
KB	knowledge base
Lisp	list programming
MINID	expert system for mineral identification
NCSL	noncommentary source lines
OPS	official production system
Pascal	a structured programming language (based on ALGOL, an algorithmic language)
Prolog	programming in logic
SCAMP	service control and monitoring prototype
STAR	software package for analyzing censored time-to-failure reliability data
STE	Statistical Test Expert
SUPER	system for prediction and evaluation of reliability

Current Expert System Tools

Expert systems are computer programs that provide expert advice and can explain their own reasoning. They consist of a *knowledge base*, which is a formal representation of expert knowledge, and an *inference engine* for processing that knowledge to provide answers to queries put to the system.

Expert systems are usually built with tools that support declarative programming as opposed to procedural programming. In *procedural programming*, the programmer instructs the computer by describing algorithms and data structures. In *declarative programming*, the programmer states specifications that the computation must satisfy.

For example, if one uses a procedural language to sort a list, the computer must be given the exact sequence of steps (the algorithm) required to reorder the list elements. With a declarative language, the same task requires the statement of conditions that must be satisfied for a list to be sorted. The system then automatically manipulates the list so that it satisfies these conditions. Less precisely, it is often said that in procedural programming, the machine must be told exactly *how* to do something, while in declarative programming, it must only be told *what* to do.[1]

In traditional procedural programming, such languages as Fortran, Cobol, C, and Lisp are used. Newer tools such as Prolog[2,3] and OPS5[4] are used for declarative programming. Declarative-programming tools have proven themselves as the tools of choice for knowledge-intensive and reasoning-intensive systems like expert systems. However, current declarative-programming tools have severe drawbacks (discussed later) that limit their use in production environments.

Expert systems and declarative programming have received enormous interest in the past several years. But there is a wide range of opinion regarding their usefulness, ranging from extreme skepticism to boundless enthusiasm. While the number of expert systems is rapidly proliferating, few have been in long-term use in real-world situations.[5,6] Thus, practical experience with expert systems and the effective scope of declarative programming is limited.

Uses of Expert System Tools and Technologies. Expert system techniques can be useful for enhancing the capabili-

ties of software tools. Current software tools tend to provide strong support for tasks that are easily specified using procedural languages (such as numerically intensive computation). These tools, however, provide much weaker support for tasks that are better specified with declarative languages (such as knowledge-intensive computation).

A good example of a knowledge-intensive task for which expert system techniques are most appropriate is the *strategy* for using a tool. This strategy concerns such questions as:

1. Can I use the tool to solve my problem?
2. How do I use the tool to solve my problem?
3. If I can't use the tool directly to solve my problem, can I use it in another way to gain insight into the problem?
4. If I can't use the tool to solve my problem, what do I do?

Each question requires expert advice, which knowledge-based systems may provide.

To address questions 1 and 4, for example, one might provide a rule-based expert user interface to the tool. Such an interface would lead a user through a consultation session asking the same questions an expert might. At the end of this session, the interface would tell the user if and how he or she might use the tool, or perhaps call the tool with the appropriate parameters. Such a knowledge-based interface has been incorporated into SUPER, an AT&T tool for hardware system reliability analysis.[7,8]

Questions 2 and 3 might be handled by several small knowledge bases used at appropriate points in a program to provide expert guidance during analysis. This strategy has been investigated using STAR, an AT&T system for life-data reliability analysis.[9]

Expert systems can also be useful for developing intelligent teaching tools.[10] Once expert knowledge has been formalized, it can be used to provide expert hints and guidance to students during problem solving, and to query students and compare their answers to those of experts. We have been exploring this technology using AVIEN (attribute-value inference engine) to construct MINID, an expert system for mineral identification developed in cooperation with the Geology Department at Franklin and Marshall College in Lancaster, Pennsylvania.

Finally, the knowledge representation techniques used in expert systems provide a way to formalize bodies of knowledge. This can help experts recognize and correct gaps and shortcomings in their own expertise.

Limitations of Current Expert-System Tools and Technologies. Many current expert-system tools have severe limitations that hinder their use in practical software-development projects. Our first concern, one shared by prominent researchers in the field,[11,12] is whether expert system technology is capable of supporting software engineering on the scale, and subject to the constraints of, typical production-software development projects. Our second concern, also shared by other researchers,[11,13] is that current expert system tools do not adequately blend facilities for declarative and procedural programming. Thus, they fail to provide a tool suitable for developers of production software. We discuss each of these concerns in turn.

Limitations for large-scale system development. Large-scale commercial and military software systems may contain several million lines of procedural code written by thousands of developers over a span of years.[14,15] Such systems typically run on a variety of general-purpose computers and are subject to severe performance constraints and memory limitations. Proven, cost-effective tools and techniques must be applied throughout the system's life cycle if the system is to be delivered on time and produced and maintained within projected costs.

To be suitable for use in developing production software, tools thus must satisfy several stringent requirements. Specifically, they must be:

- Reliable enough to be used in production software.
- Highly efficient. They can meet production performance and memory constraints.
- Portable. They can be used in software that runs on a variety of machines.
- Easy to learn and use. Developers can easily incorporate them into their everyday practice.
- Up to the standards of good software-engineering practice in terms of modularity, readability, etc.[16]

Expert system tools have emerged from a pure research environment only during the last decade. Originally, these tools supported research into expert systems themselves.

They were not written, nor were they intended to be used, for production-software development.

Because of their history, many expert-system implementation tools have serious deficiencies for production-software development. Although rapid progress is being made in correcting these deficiencies, most tools still appear to suffer from one or more problems:

- Unacceptable execution time and memory requirements. Most expert systems that are written in Lisp, Prolog, OPS5, and expert system shells based on them cannot satisfy the stringent performance constraints on production software. This problem has motivated several leading vendors of commercial expert-system tools to rewrite their tools in C language.[17]
- Portability problems, particularly the need for special hardware to achieve acceptable performance. Specialized hardware, such as Lisp machines, provides excellent support for Lisp programming and for expert systems that run in Lisp environments. But production software usually must run on many different general-purpose computers.
- Severe software-engineering problems for large knowledge bases. The largest rule-based systems in use—for example R1,[18] a system produced at Digital Equipment Corporation to configure VAX™ computers—have no more than several thousand rules. (VAX is a trademark of Digital Equipment Corporation.) Experience with this system has brought to light the difficulty of maintaining and modifying a rule base of even this scale.
- Inadequate facilities for many types of computing, particularly numerical processing and input/output processing. For example, many versions of Prolog have poor input/output facilities, and either lack floating-point operations or have inefficient ones.
- Lack of software-engineering support tools. Some environments, especially Lisp environments, provide excellent support for software development.[19] But even these environments typically fail to provide support tools for declarative programming that are comparable to those for procedural programming.
- Potentially high costs when moving to a new programming environment. These costs may include retraining programmers and developing new software-engineering support tools and function libraries. When introducing new technology, some costs are inevitable but can be minimized if the new technology is made as simple as possible and as much like the old environment as possible.

To overcome some of these difficulties, one can provide more powerful hardware to execute declarative programs. Research in this area centers on developing more powerful special-purpose hardware for declarative programming and using massively parallel machines.[20] Although this approach is promising, the portability constraints of production-software development rule out reliance on exotic hardware for most applications.

In light of these many difficulties, we conclude that, if expert system tools and techniques are to be used widely in production-software systems (which are typically large scale, and have challenging performance, reliability, and portability requirements), the bulk of these systems will have to be built using conventional procedural techniques. Such systems would be hybrid programs, consisting mostly of procedural code that calls declarative programs when necessary.

Limitations on hybrid procedural and declarative programming support. The development of production software that incorporates expert system technology must support both procedural and declarative programming because:

- Production software must be able to interface to already existing systems and libraries of code written in procedural languages.
- Given the inadequacies already cited for expert-system development tools, clearly, current large-scale production systems must be hybrid systems.

There are three approaches to combining procedural and declarative components when building hybrid systems:

1. Combine the procedural and declarative paradigms in the same language. Prolog and OPS are the premier examples in this regard.
2. Provide a way to call procedural routines from a declarative language. A good example of this approach is Borland International's Turbo Prolog™ language,[21] an implementation of Prolog for the IBM PC and compatible microcomputers that allows calls of Pascal, C,

Fortran, or assembler functions. Another such tool is the Expert System Development Environment/VM,[22] developed at IBM.

3. Provide a way to call declarative-language routines from a procedural language. An example of this approach is OPS83,[23] which allows calls in both directions between C or Fortran functions and itself. Recently, a similar implementation of OPS called C5 was completed at Bell Laboratories[24] especially for the UNIX system and C environment. Another realization of this general idea is RuleMaster2, an expert-system building tool that compiles rules written in a language called Radial into C or Fortran code.[25]

The fundamental reason that alternative 1 is poor is that the procedural and declarative-programming paradigms are radically different. Although research on the best way to combine these paradigms is currently under way, it seems clear that undisciplined mixing of constructs from both paradigms is not satisfactory for production-software development. Mixing these constructs forces the programmer to deal with more complex syntax and semantics than a purely declarative language would require. Even more important, it forces the programmer to consider flow of control while working in the declarative-programming paradigm, which is alien to the paradigm and very difficult besides. Programming environments or languages that mix the procedural and declarative-programming paradigms typically lead to code that causes unexpected behavior and is difficult to understand and maintain.

This point has been made frequently for Prolog,[11,13] which brings the `cut` operator—a procedural construct—into a programming language that is primarily declarative. Use of the `cut` operator often results in unreadable Prolog code. Similar criticisms have been made for OPS5[26] and, by inference, the entire OPS family. These difficulties are so severe that they rule out this approach for large-scale software development.

Approach 2—provide a one-way interface from a declarative language to a procedural language—is inadequate because most of a large hybrid production system must be written in a procedural language that occasionally will use knowledge-based components, rather than the other way around.

Approach 3—provide an interface from a procedural language to a declarative language (and even better, the other way as well)—is the optimal approach and the one we have pursued with AVIEN.

The CEST Strategy and Its Advantages

The basic idea of the CEST project is to provide tools for constructing high-quality expert-system components for incorporation into procedural programs. CEST will provide two classes of tools:

- A library of inference engines and support functions that are written in the C language and can be called from C programs.
- A workbench of knowledge engineering tools for building, analyzing, and maintaining rule bases.

The needs of production software for portability and high performance are met by implementing small, fast, inference engines in C. Needs for reliable, high-quality code are met by providing knowledge engineering tools.

Although we are not undertaking such work, other popular tools for building expert systems—such as C5, OPS83, and Prolog—could be incorporated into the CEST toolset as callable inference engines, if versions of these tools with appropriate interfaces to C are provided.

Advantages of This Approach. C is a small, efficient language that was originally intended for systems programming. However, libraries of functions have been supplied that effectively extend its application domain, making C a suitable tool for other types of programming as well. For example:

- The string handling library makes C a powerful language for text processing applications.
- The math library makes C suitable for numeric computation.
- `Curses`, a library of screen handling routines, enables C to provide sophisticated user interfaces for applications that require them.

Thus, a ready way to extend C to include expert system capabilities is to provide an expert-system function library.

Providing an expert-system function library of callable inference engines has advantages that help satisfy many of the requirements we listed for production-software engineering tools. First, a C function library is

easy to interface with C programs. Portability is enhanced because C is highly portable.

Reliability increases because the callable-inference-procedure approach realizes the good software-engineering practices of modularization and information hiding. An inference engine that is completely distinct from applications that call it is effectively a separate module or package whose implementation details are hidden from its users. The procedure's calling conventions and the syntax and semantics that govern the rule base specify the interface to this module. Thus, an inference engine—once completed and tested—provides reliable service to calling programs through a precisely defined interface. A good set of expert-system programming support tools should also enhance reliability by providing helpful testing and analysis tools for debugging.

Because inference engines are provided in a library, more than one engine can be made available to application programmers. This advantage has important consequences for execution efficiency, maintainability, and ease of learning and use. Generally, there is a trade-off between the efficiency of an inference engine and the power of the language in which the rules in its rule base may be expressed. For example, a small, fast, goal-driven inference engine can process rules that are expressed in a simple attribute-value language. Generally, a simpler language will also be easier to learn and use, although its expressiveness may be limited. Finally, knowledge bases that are expressed in a simpler language will be easier to maintain (if the language is expressive enough in the first place).

If several inference engines are available, the one that is most appropriate for the rule base may be called to process it. By expressing an application's rules in the simplest possible language and calling the least sophisticated (and presumably most efficient) inference engine to process them, one can achieve maximum efficiency in declarative-program processing. Such a knowledge base should also be easier to construct and maintain.

An expert-system function library and workbench effectively extends the standard software-development toolkit instead of replacing it. Consequently, developers need not relearn an entire environment and programming methodology. They need only extend their skills to include the use of another class of tools. Furthermore, because the tools we envisage are simple and often like standard tools in the UNIX system procedural-programming environment, the problems of programmer retraining and development environment retooling should be minimized.

Because a knowledge base is supplied as an argument to an inference engine, modular knowledge-base development is encouraged. The extensive set of tools that we plan for the workbench (discussed below) should also help solve the software-engineering problem for large knowledge bases.

Perhaps the greatest advantage of our approach is that it imposes a disciplined interface between procedural and declarative-programming facilities. Procedural programming needs are met by procedural programs (standard C functions). Declarative programming needs are met by declarative programs (inference engines that drive knowledge bases) that ideally contain no procedural programming constructs. Thus, we segregate procedural programs from declarative programs with no admixture of facilities from the two paradigms. The C-function call and return mechanism provides the interface between programs written in the two paradigms. The result of this strategy is that hybrid systems are more efficient, more reliable, and more maintainable.

AVIEN

As a test of this approach, we have implemented AVIEN, a backward-chaining inference engine that can be called from C programs. The inference engine uses a simple, attribute-value logic and consists of about 1600 lines of C code, which produces an object file of about 35 kbytes.

Programmers have access to the following functions for manipulating knowledge bases (Figure 1):

- `kopen` opens files of rules, facts, and attribute descriptions; parses the contents; stores them in a new knowledge base; and returns a pointer to that knowledge base. (Rules, facts, and attributes will be described shortly.)
- `infeng` is the backward-chaining inference engine. It

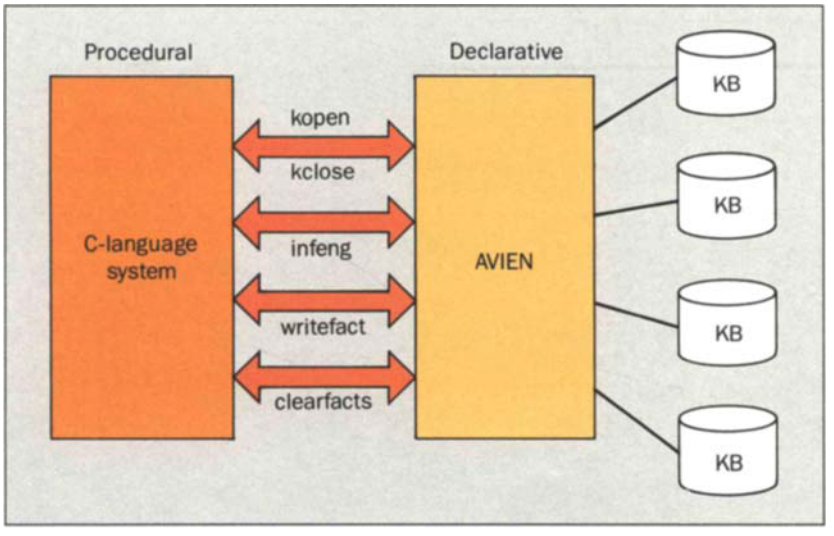

Figure 1. The C language-AVIEN interface. AVIEN provides a C-program access to knowledge bases (KBs) via C-function calls.

uses an attribute as a goal and attempts to find a value for the attribute by processing the rules, facts, and attribute descriptions in a knowledge base. `infeng` also uses an argument that indicates whether it should consult a user. `infeng` can explain its reasoning and, as an option, provide explanations of any questions it may ask a user.

- `kclose` closes a knowledge base and frees all the memory space allocated for it.
- `writefact` allows the C program that called AVIEN to write a fact directly into a knowledge base. This can be especially useful for real-time applications such as performance monitoring.
- `clearfacts` erases facts from a knowledge base.

Facts, Rules, and Attributes. An AVIEN knowledge base consists of rules and optional facts and attributes. An *attribute* (represented internally as a character string) is any property of a thing and has a *value* (again, represented as a character string). A *fact* consists of an attribute and a value connected by `is` or `are`. For example, if `circuit design` is an attribute and `correct` a value of this attribute, then

```
circuit design is correct
```

is a fact. Similarly, if `system life distributions` is an attribute and `known` is one of its values, then

```
system life distributions are known
```

is a fact.

Rules consist of one or more conjoined antecedents (the conditions) and a consequent (the conclusion). Both antecedents and consequents are facts; for example,

```
IF   level of reliability information
     is block
AND  system is multiway
THEN analysis is SUPER
```

Attribute descriptions—which are needed for interactive applications—consist of an attribute, an explanation, and a prompt. For example,

```
attr  system life distributions

translat  A cumulative distribution
   function that gives the probability
   of failure by time t

prompt  Are system life distributions
   known or unknown?
```

Here, the keyword `attr` introduces the attribute, and the information that follows `translat` explains the keyword. This explanation can be a line of text (as in the example), a path to a text file, or a path to an executable program. AVIEN uses it to respond when a user asks for an explanation of a question. The keyword `prompt` introduces the questions that AVIEN uses to prompt a user for information about an attribute when it cannot determine that information from facts or rules.

An Expert System Built Using AVIEN. Panel 2 presents a

Panel 2. Sample Execution of AVIEN

```
        Statistical Test Expert
            by W. B. Frakes
    A C Expert System Tools Application
      AT&T Bell Laboratories 1986
  Quality and Computing Theory Group

Statistical Test Expert (STE) is an expert
system designed to aid you in the selec-
tion of a test of statistical association.
STE will ask you several questions about
your data, and will then decide which test
you should use, or that it cannot provide
an answer.

How many of your variables are continuous?
1) all
2) one
3) none
4) Don't Know

? explain

A continuous variable is one that can take
on any value in a given range.

How many of your variables are continuous?
1) all
2) one
3) none
4) Don't Know

? why
          Trying to prove rule:
1) IF    number of continuous variables
         is none
2) AND   PRE interpretation is no
   THEN  measures is based on chi square
         & symmetric

How many of your variables are continuous?
1) all
2) one
3) none
4) Don't Know

? how 1

No fact or rule available to prove prem-
ise. Premise truth value must be
determined by user input

How many of your variables are continuous?
1) all
2) one
3) none
4) Don't Know

? 1

How many of your variables are artifi-
cially dichotomized?
1) two
2) one
3) none
4) Don't Know

? 3

What is the number of variables?
1) greater than two
2) two
3) Don't Know

? 2

Are ranks used or not?
1) not used
2) used
3) Don't Know

? 1

Is measure standardized?
1) standardized
2) not standardized
3) Don't Know

? 1

answer = Pearson's r
```

sample execution of AVIEN using a knowledge base about tests of statistical association. The calling C program prints the opening banner and final answer. User responses appear in bold type.

This example shows AVIEN's explanation features. When the user types **explain** in response to a question, the system gives the explanation that the knowledge base's creator provided. When the user responds with **why**, the system gives the rule it is currently trying to prove, with premises of the rule numbered. If the user types **how**—followed by a number that corresponds to a premise in the rule—and the truth value of the premise is known, the system will tell how it determined the truth value. If the truth value is not known, the system will tell how it can find out.

As an option, a full trace of the reasoning sequence is produced and stored in a file.

AVIEN Applications. So far, AVIEN has been distributed to over 60 AT&T organizations and to several universities. It is or has been incorporated into the following systems:

- SUPER is a system-reliability prediction tool[7,8] that the Quality Assurance Center at AT&T Bell Laboratories developed and supports. AVIEN was used to build an intelligent front-end to SUPER that helps users decide if and how SUPER can be used to solve reliability analysis problems.
- STAR, which the Quality Assurance Center also developed and supports, is a UNIX-system-based software package for analyzing censored time-to-failure reliability data.[9,27] AVIEN is being used to enhance the guidance strategies that STAR offers to users.
- AVIEN has been used to construct tools to assist in UNIX system administration.
- SCAMP (Service Control and Monitoring Prototype) is a prototype of a management system for complex hybrid networks. SCAMP algorithms determine if the network is in critical trouble, based on data received about out-of-service circuits and traffic load. If the network is in crisis, the AVIEN-based expert system is used to determine an ameliorative response.
- AVIEN has been used to build a grade-of-service help function for a network-customer-control prototype.

Enhancements to the Inference Engine Library

We are also examining ways to enhance the library of inference engines.

Use Other Knowledge Representations. The inference engine discussed above uses rules to represent knowledge, but rules are only one of many knowledge representations that have been proposed. Two other types of knowledge representations are semantic nets[28] and frames.[29] Others have reported[30,31] expert-system development shells that make rules, semantic nets, and frames available. We feel that this is a good approach, and although these tools are not adequate for software engineering in the UNIX system/C environment, we would like to build inference engines that provide access to at least these three representation methods.

Enhance the Inference Mechanism. We have identified the following types of inference mechanisms as candidates for addition to our function library.

- *Forward-chaining procedures* begin with everything that is known, and attempt to deduce as many other facts as possible. This strategy is often useful for problems that have many alternative solutions.
- Many models for *probabilistic reasoning* have been proposed, including Bayesian statistics and "fuzzy set" theory.[32] Several problem-solving domains involve inexact reasoning, so at least one probabilistic inference engine should be part of the projected tool set.
- Much reasoning can be captured in *propositional logic*, and inference engines for propositional logic can be very efficient. Therefore, such procedures could be a valuable addition to the library.
- For maximum expressive power, inference engines for various subsets of *first-order logic* (such as Horn clause logic) should be included in the library.

Knowledge Engineering Workbench

Any serious software-development environment must provide a good set of tools to the developer. The effort to add declarative-programming methods to the procedural programming paradigm cannot stop with the provision of callable inference engines. It must also provide tools to support the declarative programmer.

A wealth of tools has been developed to support

traditional procedural programming, the most well-known collection of which is, perhaps, the UNIX system. Unfortunately, fewer tools have been developed for declarative programming. It is not even clear which tools would be most useful. Consequently, the provision of support tools for declarative programming must, in part, be a research effort whose aim is to generate new tools and determine their usefulness.

We drew our lists of projected tools partly from tools provided in other declarative-programming environments, and partly from analogy with useful procedural programming tools.

Knowledge-Base Generation Tools. These tools help the software engineer to produce correct knowledge bases quickly and easily.

- Some success in using *syntax-directed editors* for program development has been reported for procedural languages.[33] Today, syntax-directed editors for building rule bases are available and should prove valuable to the knowledge engineer.
- For expert classification problems where the number of eventual outcomes is of reasonable size, the knowledge specification phase of an expert system project often involves drawing a decision tree. It is possible to develop a support tool that automatically translates the tree into the internal representation of rules and, as an option, produces a version of the rules for inspection.
- A *rule-induction system* automatically abstracts rules from examples that the knowledge engineer provides. Recent work on rule induction[25,34,35] suggests that such facilities may be a valuable addition to the declarative-program development toolbox.

Static Analysis Tools. Static analysis tools are used to check knowledge bases for syntactic and semantic correctness before processing. They also produce data helpful in debugging and evaluating whether a knowledge base is correct and complete.

- To help with the creation of rule bases, *consistency checkers* automatically analyze a rule base to determine if it is consistent.[36] (A rule base is *consistent* if it does not imply contradictory conclusions.) Unfortunately, consistency checking is a hard problem, both theoretically and practically. A goal of our research is to investigate further when consistency checking is feasible.
- A *rule-base summarizer* provides summary information that helps the rule base's formulator determine if the rules have been formulated properly and completely.
- Even when an adequate rule base has been formulated, it may contain unnecessary or redundant rules. A *rule-base simplifier* could root out redundancies and recommend appropriate changes to the rule base.
- Studies have questioned the usefulness of software complexity metrics, such as McCabe's metric and Halstead's metric.[37,38] But we have found certain simple measures, such as NCSL (noncommentary source lines) to be useful in estimating how fault prone and readable the code is.[39] We intend to include *metric tools* for knowledge bases that will report such data as number of rules, number of antecedents in rules, number of facts, and number of attribute records.

Dynamic Analysis Tools. Dynamic analysis tools are used to check software at run time. They help the system developer determine the software's execution efficiency, its correctness, and the extent to which it has been tested.

- *Coverage analysis tools* report the rules that have been executed, what facts have been used, what attributes used, etc.
- *Timing tools* report the execution time for an inference engine to process a query.
- *Tracing tools* provide a full trace of the inference engine's reasoning sequence for processing a query.
- Because knowledge bases lend themselves to incremental additions of knowledge, regression test sets must be provided to ensure that a modified knowledge base still functions correctly. *Regression testing tools* aid in the development and management of regression test sets.

Conclusion

Expert system techniques are a useful enhancement to standard programming development environments, but currently cannot replace them.

We have argued that expert system technologies can help solve practical software-engineering problems by providing supplements to the traditional software-engineering environment as a library of callable inference

engines and a workbench of declarative-programming tools. We have described the components of such a library and workbench and have shown its feasibility with AVIEN—a backward-chaining, attribute-value inference engine implemented as a set of C functions. AVIEN has been widely distributed within AT&T and used successfully in several projects.

We propose to continue development of the library and workbench, adding more inference engines and creating development-support tools for declarative programming.

References

1. R. Davis, "Logic Programming and Prolog: A Tutorial," *IEEE Software*, Vol. 2, No. 5, September 1985, pp. 53-62.
2. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981.
3. K. L. Clark and F. G. McCabe, *micro-Prolog: Programming in Logic*, Prentice Hall, Englewood Cliffs, New Jersey, 1984.
4. R. Brownston et al., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, Massachusetts, 1985.
5. J. McDermott, "R1: An Expert in the Computer Systems Domain," *AAAI*, Vol. 1, 1980, pp. 269-271.
6. J. R. Wright et al., "ACE: Going from Prototype to Product with an Expert System," *Proceedings ACM '84 Annual Conference, the Fifth Generation Conference*, ACM, 1984, pp 24-28.
7. S. L. Crocker et al., "SUPER: System Used for Prediction and Evaluation of Reliability," *IEEE Conference on Reliability of Computer Controlled Telecommunications Systems*, Val David, Canada, October 1985.
8. R. V. Leon and M. T. Tortorella, "The SUPER Software System for Reliability Modeling and Prediction," *1986 Joint Statistical Meetings of the American Statistical Association*, San Juan, Puerto Rico, August 1986.
9. P. Agarwala et al., "STAR: Software for The Analysis and Presentation of Reliability Data," *41st Annual Quality Congress Transactions*, American Society for Quality Control, Minneapolis, Minnesota, May 1987, pp. 264-269.
10. J. R. Anderson et al., "Intelligent Tutoring Systems," *Science*, Vol. 228, pp. 456-462.
11. J. Robinson, "Logic Programming Past, Present, Future," *New Generation Computing*, Vol. 1, 1983, pp. 107-124.
12. D. L. Parnas, "Software Aspects of Strategic Defense Systems," *American Scientist*, Vol. 73, 1985, pp. 432-440.
13. P. A. Subrahmanyam, "The 'Software Engineering' of Expert Systems: Is Prolog Appropriate," *IEEE Transactions on Software Engineering*, Vol. SE11, No. 11, November 1985, pp. 1391-1400.
14. B. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, New Jersey, 1981.
15. J. T. Beckett et al., "Methods for Managing a Large Software Project," *AT&T Technical Journal*, Vol. 65, No. 1, January/February 1986, pp. 247-271.
16. I. Sommerville, *Software Engineering*, Second Edition, Addison-Wesley, Reading, Massachusetts, 1985.
17. J. Stone, "The AAAI-86 Conference Exhibits: New Directions for Commercial AI," *AI Magazine*, Vol. 8, No. 1, Spring 1987, pp. 49-54.
18. J. McDermott et al., "R1: A Rule-Based Configurer of Computer Systems," *Artificial Intelligence*, Vol. 19, No. 1, 1982, pp. 39-88.
19. W. R. Tichy, "What Can Software Engineers Learn from Artificial Intelligence?," *IEEE Computer*, Vol. 20, No. 11, November 1987, pp. 43-54.
20. K. Hwang, J. Ghosh, and R. Chowkwanyun, "Computer Architectures for Artificial Intelligence," *IEEE Computer*, Vol. 20, No. 1, January 1987, pp. 19-27.
21. *Turbo Prolog Owner's Handbook*, Borland International, Inc., Scott's Valley, California, 1986.
22. *Expert System Development Environment/VM Reference Manual*, IBM Corporation, Irving, Texas, 1985.
23. C. L. Forgy, *The OPS83 Report*, Technical Report CMU-CS-84-133, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1984.
24. G. Vesonder, "Rule-Based Programming in the UNIX® System," *AT&T Technical Journal*, Vol. 67, No. 1, January/February 1988, pp. 69-80.
25. D. Michie et al., "RuleMaster: A Second Generation Knowledge Engineering Facility," *Proceedings of the First Conference on Artificial Intelligence Applications*, Denver, Colorado, 1984.
26. R. E. Cooley, "Production Systems for Expert Systems Shells," *Proceedings of the ESD/SMI Expert Systems Conference and Exposition for Advanced Manufacturing Technology*, Dearborn, Michigan, June 1987, pp. 223-228.
27. W. B. Frakes and C. J. Fox, "Development of Expert Systems for Quality and Reliability Using an Expert System Function Library and Workbench for UNIX/C," *41st Annual Quality Conference Transactions*, American Society for Quality Control, Minneapolis, Minnesota, May 1987, pp. 253-263.
28. W. A. Woods, "What's in a Link: Foundations for Semantic Networks," *Representation and Understanding*, D. G. Bobrow and A. Collins (eds.), Academic Press, New York, 1975.
29. P. H. Winston and B. K. P. Horn, *Lisp*, Addison-Wesley, Reading, Massachusetts, 1981.
30. N. Lee and J. Roach, "GUESS/1: A General-Purpose Expert System Shell," *Expert Systems in Government Symposium*, The MITRE Corporation and IEEE Computer Society, McLean, Virginia, 1985.
31. P. Hammond, "APES: A User Manual," Report No. 82/9, Department of Computing, Imperial College, London, 1982.
32. B. G. Buchanan and R. O. Duda, "Principles of Rule-Based Expert Systems," Technical Report No. HPP-82-14, Stanford

University Heuristic Programming Project, Palo Alto, California, August 1982.
33. T. Teitelman and T. Reps, "CPS—The Cornell Program Synthesizer," *Communications of the ACM*, Vol. 24. No. 9, September 1981, pp. 563-573.
34. R. S. Michalski, "A Theory and Methodology of Inductive Learning," *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (eds.), Tioga Publishing Co., Tioga, New York, 1983, pp. 83-134.
35. R. S. Michalski and R. L. Chilausky, "Learning By Being Told and Learning from Example: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *Policy Analysis and Information Systems*, Vol. 4, No. 2, February 1980, pp. 125-160.
36. T. A. Nguyen et al., "Knowledge Base Verification," *AI Magazine*, Vol. 8, No. 2, Summer 1987, pp. 69-76.
37. M. Evangelist, "Software Complexity Metric Sensitivity to Program Structuring Rules," *The Journal of Systems and Software*, Vol. 3, No. 3, March 1983, pp. 231-243.
38. S. Crawford et al., "An Analysis of Static Metrics and Faults in C Software," *Journal of Systems and Software*, Vol. 5, No. 1, January 1985, pp. 37-48.
39. W. B. Frakes and C. J. Fox, "An Approach to Integrating Expert Components into Production Software," *Proceedings of ACM/IEEE Fall Joint Computer Conference*, Dallas, Texas, 1987.

Biographies (continued)

and an M.S. in statistics and Ph.D. in information science from Syracuse University.

(Manuscript received July 6, 1987)