Perspectives on Artificial Intelligence Programming

DANIEL G. BOBROW AND MARK J. STEFIK

Programs are judged not only by whether they faithfully carry out the intended processing but also by whether they are understandable and easily changed. Programming systems for artificial intelligence applications use specialized languages, environments, and knowledge-based tools to reduce the complexity of the programming task. Language styles based on procedures, objects, logic, rules, and constraints reflect different models for organizing programs and facilitate program evolution and understandability. To make programming easier, multiple styles can be integrated as sublanguages in a programming environment. Programming environments provide tools that analyze programs and create informative displays of their structure. Programs can be modified by direct interaction with these displays. These tools and languages are helping computer scientists to regain a sense of control over systems that have become increasingly complex.

PEOPLE WHO DEVELOP PROGRAMMING SYSTEMS NEED TO organize them in ways that make them comprehensible to other people and easy to modify. We will be concerned mainly with the practice of artificial intelligence (AI) programming and about the arguments and tensions that have shaped the current state of the art. An intelligent system should embody and apply information about itself so that it can assist in its own continuing development.

The programming culture of the AI community has a somewhat different emphasis than most "production" programming. The AI programming process is not a sequential progression from specification to implementation, testing, and release. Instead, an exploratory approach is used in which specification and implementation evolve together as the problem is understood and tested. Another difference is the frequent design of experimental specialized sublanguages that make it easier to express solutions to the problems being attacked.

List processing (*1*) is fundamental to AI programming (*2*). As an example, the list

(IBM (A-Kind-Of ComputerCompany)(Headquarters NYC))

can be used to represent relations between the thing represented by the symbol IBM and other things represented by the symbols ComputerCompany or NYC. Manipulations of these list structures can deduce implicit relations (*3*), for example, that IBM produces computers (because it is A-Kind-Of ComputerCompany). Programs can use lists to build structures of unpredictable sizes and shapes during execution without predetermined or artificial limits.

List structures are also used to represent programs in AI systems, and hence they are often used to build tools to infer implicit features about programs themselves. This contrasts with typical programming systems that deal with programs as a sequence of characters; program changes are made by adding and deleting characters. Higher level organizations of programs, such as the module boundaries and the calls between packages and so on, are parsed from these characters but made available in very limited ways to the users. AI systems provide users with interactive displays that describe systems in these terms. A user can understand a system and change it directly through these displays rather than indirectly by manipulation of text.

A programming style is a way of organizing programs on the basis of some conceptual model of programming and an appropriate language to make programs written in the style clear. We will describe styles organized around procedures, objects, logic, rules, and constraints. Each style is a specialized language or sublanguage that shapes the organization of programs written in that style.

Different styles differ substantially in what can be stated concisely. Significant appeal arises from what does not have to be stated. By eliminating redundancy, the intent of the code can be more easily understood. This is an important virtue of, for example, automatic storage management facilities that allow omission of code for freeing storage. Different styles facilitate different kinds of program change that ensure appropriate properties of the program remain invariant. Experiments have led to systems in which a number of styles are integrated and to others in which one style dominates.

Programming Styles

Procedure-oriented programming. In this style, subroutines and data structures are the two (separate) primitive elements. Subroutine calls are the primary mechanism for program composition. Subroutines have the property that they carry out the same algorithm when called from different places. Adding a line to a calling program, and thus changing the position of the call to a subroutine, does not change the algorithm executed by the subroutine.

Data structure declarations allow programs to reference parts of a complex data structure by name rather than, for example, by index position in an array. In the declaration, a programmer can specify once the method for looking up the named substructure rather than specifying it in every place that the structure is referenced in the procedures. To change the lookup process, a programmer need only change the specification. AI systems provide automatic storage management facilities that ensure that any data structure no longer referenced directly or indirectly by the program is reclaimed. This avoids problems that occur in systems where a programmer must

Daniel G. Bobrow is a research fellow and Mark J. Stefik is a principal scientist in the Intelligent Systems Laboratory of the Xerox Palo Alto Research Center, Palo Alto, CA 94304.

take responsibility for storage management, such as failure to return unused storage or inappropriate deallocation of referenced storage.

Although programmers do not usually think about the invariants associated with subroutine call, data access, or storage management, they are essential for the composition of large programs. They confine and control the effects of change. To the extent that common changes are local, a language provides insulation from change-induced bugs.

Some languages augment the notion of subroutine with features intended to support better the sharing of code by several people. Modula-2 (*4*) and Ada (*5*), for example, provide mechanisms for defining modules—collections of related procedures and structure definitions—and interfaces—descriptions of elements of a module that may be used from outside that module. The interfaces are intended to minimize interference as people change different parts of a system.

The interface definition provides information about module data types and procedure requirements that can be used for module optimization. An implementation of a module is free to change any features not "advertised" in the interface. With these restrictions, independently developed, debugged, and compiled modules can be loaded together. The interface definition is the defined narrow pathway of interaction.

Object-oriented programming. This style (*6, 7*) has often been advocated for simulation programs, systems programming, graphics, and AI programming. Although there are variations in exactly what is meant by object-oriented programming (*8*), in all these languages there are objects that combine state and behavior.

There are three major ideas in object-oriented programming: (i) objects are defined in terms of classes that determine their structure and behavior; (ii) behavior is invoked by sending a message to an object; and (iii) descriptions of objects may be inherited from more general classes. Uniform use of objects contrasts with the distribution of information into separate procedures and data in procedural programming.

A message to an object contains a name for a behavior (often called its selector) and some other parameters. For example, in a traffic simulation we could have classes of vehicles such as Car and Truck. To cause Car-1, an instance of the class Car, to move in the simulation, a Loops (*9*) program would send a "Move" message to Car-1:

```
(send Car-1 Move 400 50)
```

Associated with the class Car is a particular method for Move that is run for this invocation. The class Truck has a different method, since trucks must obey different traffic rules and consume different fuel. The "message passing" style allows each class to implement its response to a message in its own way. These methods can be changed independently. In contrast, procedure-oriented programming would require that all the variations of Move be incorporated into the single procedure that implements Move.

Classes can inherit description from other classes. PoliceCar can be defined as a specialization of the class Car. Then PoliceCar has all of the structure and behavior of Car except that which is explicitly overridden or added in PoliceCar. For example, PoliceCar can add a two-way radio and a method for Move that can exceed the speed limit or interact with traffic light control. Such inheritance of a specialized class from its "superclass" reduces the need to specify redundant information and simplifies updating and modification, since information can be entered and changed in one place.

Specialization and message sending synergize to support program extensions that preserve important invariants. For example, splitting a class, renaming a class, or adding a new class does not affect simple message sending unless a new method is introduced. Instances of a specialized class follow exactly the same protocols as a superclass until local specialized methods are defined. Similarly, deleting a class does not affect message sending if the deleted class does not have a local method involved in the protocol.

Changes to the inheritance network are common in program reorganization. Programmers often create new classes and reorganize their classes as they understand the opportunities for factoring parts of their programs. Together, message sending and specialization provide a robust framework for extending and modifying programs.

Access-oriented programming. In object-oriented programming, when an object is sent a message it may change the values of some variables that make up its internal state. In access-oriented programming (*10*), when an object changes the value of a variable a message may be sent to another object as a side effect if the value associated with that variable is an "annotated value." In terms of actions and side effects this is dual to object-oriented programming. The annotated value is a specialized object and can contain state other than the value.

Access-oriented programs are factored into parts that compute and parts that monitor the computations. For example, suppose one were to build a traffic simulation program with an interactive display showing the state of the simulation. By dividing it into a simulator and a display-controller, one can separate programming concerns.

The simulator represents the dynamics of traffic. It has objects for such things as automobiles, trucks, roads, and traffic lights. These objects exchange messages to simulate traffic interactions. For example, when a traffic light object turns green, it sends messages to start traffic moving.

The display-controller has objects representing images of the traffic and provides an interactive user interface for scaling and shifting the views. It has methods for presenting graphics information. The simulator and the display-controller can be developed separately, provided that there is agreement on the structure of the simulation objects.

Access-oriented programming provides the "glue" for connecting the simulator and display-controller. The process of connection is dynamic and reversible. When a user tells the display-controller to change the views, it can make and break connections to the simulator as needed for its monitoring. Thus at one time a user can monitor the simulation as if looking at a map of the town and at another time as if looking at the instrument panel of a particular car. For the former, active values attached to the positions of each vehicle are used to update the display; for the latter, probes on the speed of the auto and level of the gas tank can update pictorial gauges. Attaching such gauges does not change the behavior of the program being monitored.

Property annotations in annotated values can be used to store useful but subsidiary quantities. Some systems store a measure of the certainty of that value being correct. A reasoning system can store this annotation without having to change the structure of the represented object. Other values and rules used in computing this value could be stored as annotations, as in truth maintenance systems (*11*). Annotations also provide a place for documentation for human readability of data structures.

Access-oriented programming supports several invariants under program change. Annotations can be added to programs without causing them to stop working. Annotated values are invisible to programs that are not looking for them. They can be added to data that are already annotated. The same invariants hold for recursive annotated values, that is, for descriptions of descriptions. Nested active values enable multiple independent side effects on variable access.

Logic programming. There are both declarative and procedural interpretations of logic statements (*12*). The simplest statements in Prolog, the most popular logic-based language, is a statement of a relation; for example:

Brother(*Danny, Rusty*)

As a declarative statement, it has a truth value; procedurally it is a request to check the truth of the statement. Prolog uses a database of facts entered by statements of the form

Assert(Brother(*Danny, Rusty*))

to store this relation in a database. A query Brother(*Danny, X*) searches the database and returns $X = Rusty$. The same elementary fact could also be used to answer the query Brother(*X, Rusty*).

In addition to simple statements and queries, Prolog programs consist of sets of statements stored in the database. Each statement has the form

consequent :- antecedent-1, . . . , antecedent-*n*

This is read declaratively as, consequent is true if (:-) each of antecedent-1 to antecedent-*n* is true. In a procedural reading, the consequent is taken as a goal to be achieved, and the antecedents are subgoals to be tried in order. The declarative interpretation of the statement,

Uncle(X, Y) :- Brother(X, Z), Father(Z, Y)

is "X is an Uncle of Y if there is some Z, such that X is a Brother of Z, and Z is the Father of Y."

This statement can be read as directions for achieving a goal Uncle(X, Y): first find a Z such that Brother (X, Z), and then prove that Z is a Father of Y. To verify the truth of Uncle(*Danny, Johanna*), the system could find Brother(*Danny, Rusty*) and Father(*Rusty, Johanna*).

Unlike ordinary procedures, any number of the inputs to a program can be left unspecified. The system searches for one (or upon request, more) bindings of the input parameters that make the consequent true. By repeated application of the Uncle rule, starting with Uncle(*Danny, X*), the system will find all the nephews and nieces of *Danny*; Uncle(*X, Johanna*) can be used to find all the uncles of *Johanna*.

Since the given Uncle rule does not completely specify the Uncle relation, a second rule can be added after the first:

Uncle(X, Y) :- Brother(X, Z), Mother(Z, Y)

After searching the database (or using other rules) with the first rule, Prolog would then "backtrack" to use the second rule. In general, any number of ordered rules can be used to specify a relation. An advantage of using Prolog is that new rules can be easily added to modify the system behavior. The system will perform exhaustive search with all the rules, and with all clauses from the database, to find appropriate bindings for input parameters that are unspecified.

One of the important features of logic programming is that is separates the idea of goals from the statements of how to satisfy them. This reification of goals, rather than explicit calls to particular methods for achieving them (subroutines), allows new methods to be added without the need to change the goal statements. This is an advantage in the incremental development of a system, but it may make program behavior hard to understand.

Automatic search through the database of rules rather than explicit control has another disadvantage. Some obvious and innocent-looking rules, such as that expressing the commutativity of the Brother relation,

Brother(X, Y) :- Brother(Y, X)

can cause the system to go into an infinite loop. The problem of search control is an area of active research in the logic programming community.

Because Prolog provides simple rules with clear declarative semantics and a convenient interpretation for database search, it was chosen as the starting point for the fifth-generation computer project in Japan (*13*).

Rule-based programming. The widespread use of the term rule-based programming belies the considerable diversity in what it is used to mean (*14, 15*). Rule languages are used to support the building of knowledge bases. Rule environments include tools for generating explanations of program behavior, tools for answering questions, and tools for acquiring and integrating new rules into a program.

Rule languages use if-then statements as shown in this rule from Mycin (*16*):

If the Gram stain of the organism is Gram-negative,
the morphology of the organism is rod, and
the aerobicity of the organism is anaerobic,
then there is suggestive evidence (0.7) that the identity of the organism is bacteriodes.

This rule can be used by reasoning forward from laboratory tests to accumulate evidence about the identity of a disease organism. It is also used to reason backward from a goal of finding the identity of the invading organism to determine what tests should be run. The (0.7) in the above rule is a weighting for the evidence mentioned in the preconditions of the rule. Extensive work has been done in the context of rule-based systems on techniques for combining multiple pieces of evidence (*17*). Uncertain information is not usually handled in logic-based systems.

Rule-based systems provide explanations of results to users by keeping track of which rules were invoked in a consultation. After a consultation they can explain the reasoning the system used. A fundamental assumption in these systems is that the display of rules applied is a reasonable explanation of system behavior. Explanations do not reflect assumptions underlying rules, causal mechanisms, or the current knowledge of the user.

For small tasks, rules are viewed as independent, and getting them right is easy because rules are small and manageable. For large tasks, the interactions of rules must be considered. Some rule languages organize rules into hierarchical rule sets that describe how the rules are to be applied. Some provide problem-solving frameworks in which rules can be organized. For example, each subtask in a network can carry its own relevant rules, as in Pride, an expert system for aiding in a mechanical design (*18*). In Blackboards (*19*), rules are attached to nodes of a general problem-solving model. In structured systems, the programming task of adding new rules also requires deciding where to put them.

In summary, the available rule languages are important in narrow, carefully chosen applications, often for expert systems. Their fundamental strength—the construction in terms of independent individual rules—is also their limiting factor.

Constraint-oriented programming. The idea of developing programming languages around the concept of constraint satisfaction has appealed to computer scientists for years (*20*). The idea is that a programmer need only declare certain relations among program variables without saying precisely how they should be achieved. The details of the computation can then be figured out by the system by means of implicit, automatic constraint satisfaction techniques.

For example, the equation $x + y = 5$ can be viewed as a constraint on possible values for the variables x and y. If numeric values for x and y are given, we can substitute those values into the equation and determine whether the constraint is satisfied. In this example, a

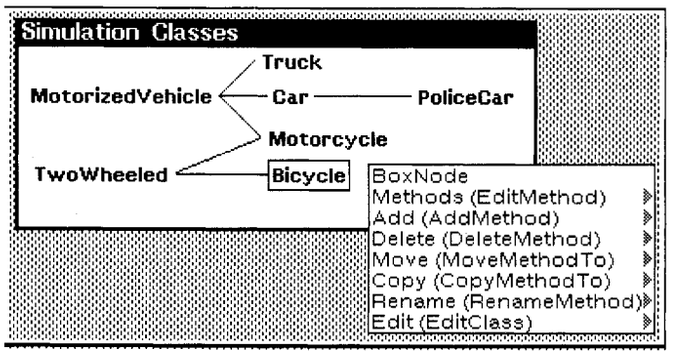

Fig. 1. A browser, showing the inheritance of classes for the simulation. On the right is an interactive menu for some of the operations that can be done by pointing to elements of this browser. The Add item on the menu allows addition of structure, methods, or classes. As indicated, the default operation is to add a method. Such browsers provide a simple way of specifying common changes and maintain an up-to-date display of the program structure throughout the process.

numeric value for either variable together with the constraint is enough to determine the value of the other variable.

The most widely used and practical systems based on constraints are the spread-sheet programs that have been popular on personal computers. Spread-sheet programs provide a matrix of rows and columns for organizing values and constraints among them. For example, in a rental income application, columns of the matrix could correspond to months of a year and rows could correspond to income and categories of expense. Constraints connect dependent elements, so that total monthly expense is maintained as the sum of the individual expenses in the same column. When monthly rental rate, taxes, and utilities variables are filled in for the first month, constraints in a spread-sheet fill in as much of the table as possible, including various subtotals.

The constraints in a spread-sheet program differentiate between dependent and independent variables. When the independent variables are filled in, values for dependent variables are updated immediately. Reasoning in both directions is provided in some applications, for example, in graphics and simulation (*21*).

Constraint satisfaction systems have always been specialized to exploit the nature of the particular kinds of constraints and hence have not been considered general purpose. Nonetheless, there are several research efforts aimed at increasing the breadth of applicability of constraint languages for specifying the behavior of computers (*22*).

Use of Multiple Paradigms

Programming language development has often consisted of the honing of a particular paradigm for organizing programs (*23*). Advocates of multiple styles in a single system (*24, 25*) argue that, just as there are many tools in a carpenter's toolbox, each specialized to its purpose, there should be many tools in the programmer's kit. One should not be forced to pry up nails with a screwdriver. However, use of multiple paradigms does involve an additional cost of learning more than one style, and programs may be required to transform between different representations of the same information chosen to optimize processing within a style.

When a problem does not fit well in a style, resulting programs may be both awkward and long. For large applications, the various costs for using a particular style can vary across parts of the program. In addition to the cost of the initial writing of the program and the cost of running the program, the costs of debugging and change as the program and its specifications evolve must be considered. The total cost of a system can be lower when more than one style is used. For example, many expert systems are developed in which object-oriented programming is used for representing the basic concepts, rules are used to specify the inferences, access-oriented programming is used to drive the graphics display, and procedures are used for the overall control structure.

Sometimes the search for integration can lead to a language that gracefully subsumes the different styles. This is illustrated in the deep integration of procedure-oriented programming and object-oriented programming in CommonLoops (*26*). In ordinary procedure calls, the code to carry out an operation is looked up by using only the name of the procedure. In object-oriented programming, the code lookup process for message sending uses both an operation name (the selector) and also the class of the first argument.

Procedure call and message sending are generalized in CommonLoops, so that code lookup uses the selector and the types of as many arguments as desired (multimethods). Thus, CommonLoops does more than just provide both message sending and procedure call. The integration yields a continuum of method definitions from simple procedures to methods with many arguments whose types are specified. The familiar methods of object-oriented programming fall out as a special case where the type (class) of only the first argument is used. A programmer using code developed by others need not be aware of whether there are multiple implementations that depend on the types of multiple arguments.

Computer scientists are just beginning to develop examples of the integration of styles in hybrid or integrated languages and criteria for judging them (*25*). Different programming languages are no longer just focusing on a particular style; styles now coexist and are beginning to evolve together (*27*).

Programming Environments

Programming languages reduce the complexity of programming by simplifying the expression of instructions. Programming environments are the set of tools used to build, change, and debug programs. Operating systems have played this role, but now specialized environments (*28*) that know about the language and program structure reduce complexity by taking some responsibility for managing changes to programs.

AI programming environments provide tools that analyze program structure and create informative displays that help programmers to develop mental models of the systems. They also provide simplified means for specifying changes to a program that free a programmer from specifying many of the details. Tools may also be used to compensate for a particular distribution of information imposed by one or more styles of programming. These tools are particularly important in the exploratory programming style (*29*) used in AI, where the specifications for a task are developed as parts of it are implemented.

Understanding and changing the static structure of a program. Traditionally, the main descriptions of programs available to programmers have been the text of program instructions. This is useful when a program fits on a few pages, but stacks of program listings are inadequate for visualizing large programs. Nor is the situation much improved by computerizing the same view with text editors and window systems. Text editors do not provide a flexible overview and are of limited use in making many important kinds of systematic changes.

The primary struggle is often to simplify the organization of a system. Simplification may require exploring and changing the

boundaries between and within subsystems. It is now possible to create automatically more informative views of programs that reflect the kinds of questions that programmers ask when they are modifying or trying to understand a large system.

For procedure-oriented programming, tools can display interactive graphs that show where procedures and variables are defined and used (*30*). In object-oriented programming, changes to the inheritance network are common in program reorganization. An interactive class browser such as that shown in Fig. 1 can make it easy to add, delete, and rename classes and methods, examine documentation, trace the inheritance of particular methods or variables, and move definitions of methods and variables in the inheritance lattice.

Understanding and changing the dynamic structure of a program. Testing, debugging, and performance tuning are all important parts of the task of programming. In the current state of the art, these tasks cannot be done practically by an analysis of static program structure, and so programming environments provide interactive tools for them.

Conventional programming practice has long included means for tracing the execution of programs to aid in debugging. In the simplest case, tracing is achieved by inserting statements into a program to cause printing when various parts of a program are activated and to print out indications of the internal state. Other capabilities allow interruption of program execution when certain conditions arise, such as when a particular procedure is called, a variable is accessed, or a value is set that is outside a specified range.

An improvement on tracing is the use of gauges, as shown in Fig. 2. Several systems that support access-oriented programming provide a suite of gauges that can be attached to the variables of running programs to display the monitored values. Attaching a gauge to a program variable is analogous to attaching a voltmeter to a circuit. The gauge does not interfere with the operation of the program, and it is not necessary to search through a long listing to find values for variables. A programmer can create an array of gauges on the display and watch them while the program runs.

Modern environments let a programmer examine the state of a computation when a program is interrupted, either by an error or by a user request. If a bug is found and a fix made, the user can back up from a nested computation to a call that invoked it and try the computation again from that point. Some systems can also run a program at slow speed or one step at a time. The ability to interrupt a program's execution to make changes can make a dramatic difference in the overall productivity of programming because this enables a programmer to identify and correct several errors in a single short session.

Many of these interactive capabilities for controlling execution are not new ideas. Some of them have been available in Basic and Lisp systems for several years. They are mentioned because they are less common in production programming environments for large systems, they are important for incremental debugging, and they are usefully combined with tools for analyzing and modifying program structure.

AI programming environments tend to do late-binding by default. This means that they usually provide the flexibility for programmers to change a running program without losing the state of the computation. Programmers are concerned about the time it takes to complete a cycle of revision: discover a problem, find a bug, make a revision, and test again. Late-binding systems tend to speed up this cycle.

Late-binding can slow program execution. Optimization facilities in AI systems allow compilation of efficient programs before release for wide use. Exploratory programming encourages a style of programming in which exploration is followed by analysis, which is

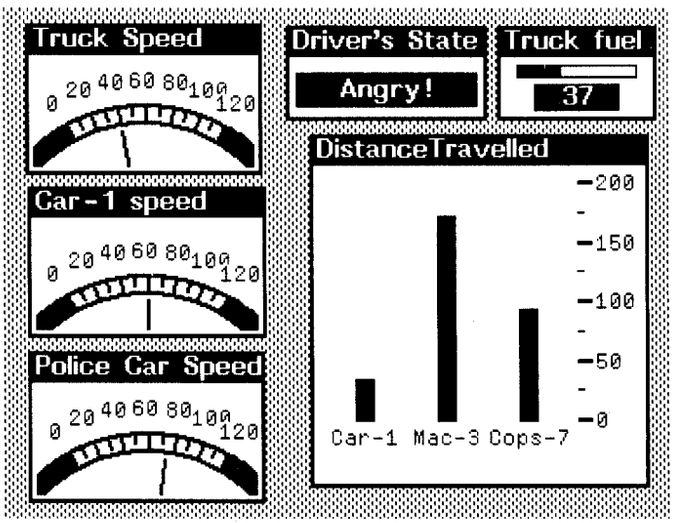

Fig. 2. Gauges, which can be attached to any object variable at any time to view the changing value of a variable. A set of attached gauges allows a simultaneous view of the dynamic state of a program.

then followed by optimization. In this approach, optimization is focused where it is needed. This leads to effective optimization guided by real measurements rather than being based on preconceptions (often misconceptions) of system designers. A program's performance can be largely determined by the performance profile of underlying system facilities, whose performance on particular cases may not be known ahead of time.

A graphical view of timing analysis, as shown in Fig. 3, can be useful for understanding the incremental and integrated time spent in any part of the system during a particular computation.

Narrow Knowledge-Based Systems

Work on programming environments can be understood as an application of computer technology to the task of writing and maintaining programs. The same theme can be seen in the work on compilers. Compilers convert high-level language descriptions into specific instructions. They use their knowledge of machine architecture to yield efficient implementations. Compilers, like programming environments, are intended to be used for all kinds of programming tasks.

An important strategy for making tools that can give more comprehensive kinds of assistance is to incorporate specific knowledge into them. In the 1960's work on compilers was sometimes called automatic programming. Current research on automatic programming (*31, 32*) follows this direction of incorporating increasing amounts of knowledge about programming. The most successful uses of automatic programming are even more narrowly focused; these are the application generators now used commercially for creating specially tailored systems for business applications, such as accounting and inventory.

The same trend toward knowledge-intensive systems can be seen in the creation of so-called shells for expert systems. An expert system shell is a specialized sublanguage and environment designed to support a set of closely related applications. Shells are an intermediate point between specific applications and general-purpose "knowledge-engineering" environments. Shells can be built for such applications as planning, scheduling, and a variety of specialized office tasks.

Shells have four things that general programming tools do not:

prepackaged representations for important concepts, inference and representation tools tuned for efficient and perspicuous use in the applications, specialized user interfaces, and generic knowledge about the application. For example, a shell for a planning application could have representations for modeling goals and agents. Its specialized knowledge could include rules, such as one specifying that an agent can be only at one place at a time. It would have generic categories for things such as time, tasks, serially reusable resources (such as a room), and interfaces for interacting with alternative plans.

These knowledge-intensive systems blur the boundaries between environments and languages, since they combine features of both. Domain-specific knowledge enables a system to take on more of the responsibility for checking and installing changes. Domain-specific knowledge also enables systems to provide specialized user interfaces that are intended to be closer to the concepts of the application. General-purpose environments have broad applicability; specialized systems can do more in a narrower domain.

Directions and Themes Revisited

The developments we have discussed in programming languages, environments, and knowledge systems provide different perspectives on what a program is. Conventionally, a program is a set of instructions for a machine that specifies how information is to be processed. Programming is the process of translating user intentions and requirements into a formal language understandable by a computer.

Variations in programming languages determine what can be stated concisely, what must be stated in multiple places, and what need not be stated in a program. Objects, rules, procedures, constraints, and other programming concepts make different tradeoffs in the way that they organize information. If a programming language allows one to write procedures but not constraints, then expressing desired relations among variables requires having statements in all those parts of the program that can potentially change the values of the variables. Inheritance allows structural description and methods to be shared by classes without redundant specification.

When a system makes directly manipulatable the concepts of an application, programs become more understandable. For example, some bookkeeping and accounting concepts are represented and manipulated directly in spread-sheet programs. While the state of the art has no dependable cognitive metric for how much this helps, the issue is a recurring theme in the design of languages and knowledge-based systems. It has to do with reducing the levels of abstraction that must be penetrated to understand system behavior.

In contrast, many of the programs for modern physics experiments have become large and perhaps unmanageable. There are so many levels of mathematical technique and abstraction between the terminology of physics and the text of the programs that they have become unwieldy and hard to understand. Indeed, one important role for AI systems is as an impedance matcher, or natural bridge, between mental concepts and program symbols. The Sophie system (*33*) is an example of a program that, among other things, provides an interface between descriptions of circuits and Spice, the underlying simulation program for modeling circuit behavior. Hybrids like this suggest ways of using programs in different contexts, thus "preserving programming capital."

Programming environments provide us with another answer to what a program is. Environments determine what a programmer sees when writing or modifying a program. They include different kinds of interactive browsers, as seen in Figs. 1 to 3, that provide different views of a program based on automatic analysis. These visualizations are designed to help programmers gain perspective on their programs, and in doing this they blur the boundary between language and environment.

As browsers are increasingly used for understanding and changing systems, they displace program listings. Ultimately they are more powerful (because they are active), can employ specialized knowledge, and can provide alternative views. Today's programs are parts of larger complex systems, and the main activity of programming has moved from the origination of new programs to the modification of existing ones (*34*). Programs are increasingly judged not only by whether they faithfully carry out the intended processing but also

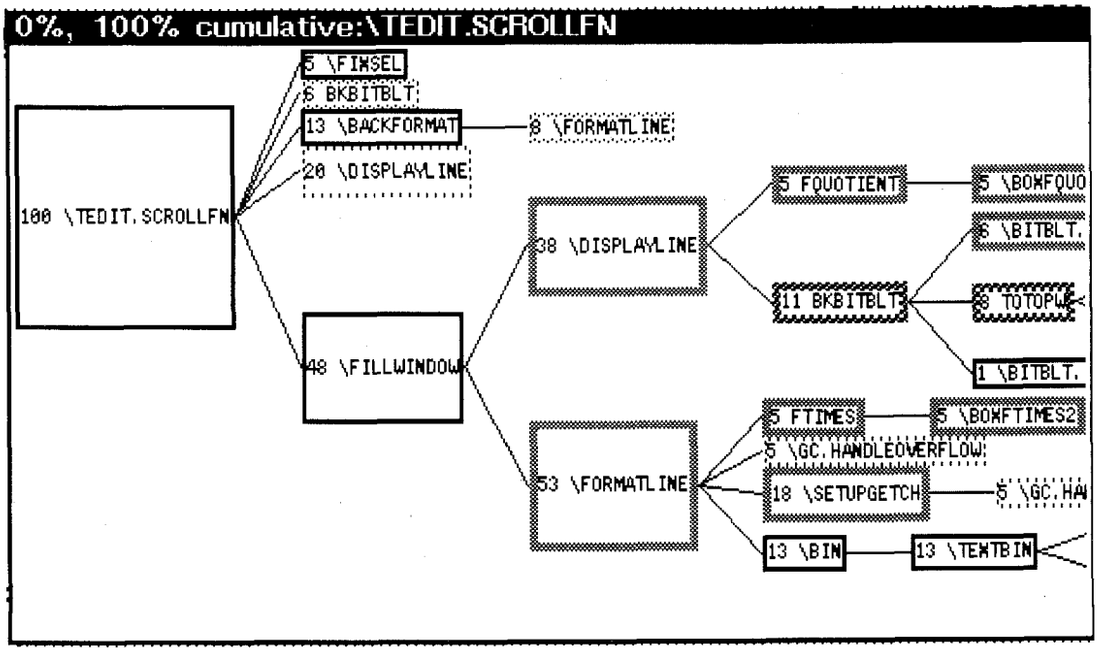

Fig. 3. Spy timing-analysis tree in Interlisp-D (*35*). The height of each box is proportional to the fraction of the time spent in the routine. Large boxes are associated with potential candidates for optimization. The border of each box is used to indicate modes, such as "time includes called subroutines," "appears elsewhere on display," and so forth.

by whether they are understandable and easily changed. Thus computer tools that bring computational leverage to programming are helping computer scientists to regain a sense of control over systems that have become increasingly complex.

REFERENCES AND NOTES

1. J. McCarthy, *Commun. ACM* **3**, 185 (1960).
2. E. Charniak, C. Riesbeck, D. McDermott, *Artificial Intelligence Programming* (Erlbaum, Hillsdale, NJ, 1980).
3. R. Davis, *Science* **231**, 957 (1986).
4. N. Wirth, *Programming in Modula-2* (Springer-Verlag, New York, 1985).
5. B. Wichman, *Commun. ACM* **27**, 98 (1984).
6. G. Birtwistle, O. Dahl, B. Myhrhaug, K. Nygaard, *Simula Begin* (Auerbach, Philadelphia, 1973).
7. A. Goldberg and D. Robson, *Smalltalk-80, The Language and Its Implementation* (Addison-Wesley, Reading, MA, 1983).
8. M. Stefik and D. G. Bobrow, *AI Magazine* **6**, 40 (1985).
9. D. G. Bobrow and M. J. Stefik, *The Loops Manual* (Xerox Corporation, Palo Alto, CA, 1983).
10. M. Stefik, D. G. Bobrow, K. Kahn, *IEEE Software* **3**, 10 (1986).
11. J. Doyle, *Artif. Intell.* **12**, 231 (1979).
12. R. A. Kowalski, *Commun. ACM* **22**, 424 (1979).
13. T. Moto-oka, Ed., *Fifth Generation Computer Systems* (Elsevier/North-Holland, Amsterdam, 1982).
14. R. Davis and J. King, in *Machine Intelligence*, E. Elcock and D. Michie, Eds. (Wiley, New York, 1976), vol. 8, pp. 300–332.
15. D. Waterman and F. Hayes-Roth, Eds., *Pattern-Directed Inference Systems* (Academic Press, New York, 1978).
16. B. G. Buchanan and E. H. Shortliffe, *Rule-Based Expert Programs: The MYCIN Experiments of the Stanford Heuristic Programming Project* (Addison-Wesley, Reading, MA, 1984).
17. J. Gordon and E. H. Shortliffe, *Artificial Intelligence* **26**, 323 (1985).
18. S. Mittal, C. L. Dym, M. Morjaria, in *Applications of Knowledge-Based Systems to Engineering Analysis and Design*, C. L. Dym, Ed. (American Society of Mechanical Engineers, New York, 1985), p. 99.
19. B. Hayes-Roth, *Artif. Intell.* **26**, 251 (1985).
20. I. E. Sutherland, thesis, Massachusetts Institute of Technology, Cambridge (1963).
21. A. Borning, *ACM TOPLAS* **3**, 353 (1981).
22. G. Steele, thesis, Massachusetts Institute of Technology, Cambridge (1980).
23. H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs* (Massachusetts Institute of Technology Press, Cambridge, 1985).
24. R. Fikes and T. Kehler, *Commun. ACM* **28**, 904 (1985).
25. D. G. Bobrow, *IEEE Trans. Software Eng.* **SE:11**, 10 (1985).
26. D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, *CommonLoops, A Graceful Merger of Common Lisp and Object-Oriented Programming* (Xerox Corporation, Palo Alto, CA, 1985).
27. B. Hailpern, *IEEE Software* **3**, 6 (1986).
28. A. Goldberg, in *Interactive Programming Environments*, D. Barstow, H. Shrobe, E. Sandewall, Eds. (McGraw-Hill, New York, 1984), p. 141.
29. B. Sheil, *ibid.*, p. 19.
30. W. Teitelman and L. Masinter, *ibid.*, p. 83.
31. D. Barstow, *AAAI Magazine* **5**, 5 (1984).
32. C. Rich and H. Shrobe, in *Interactive Programming Environments*, D. Barstow, H. Shrobe, E. Sandewall, Eds. (McGraw-Hill, New York, 1984), p. 443.
33. J. S. Brown, R. Burton, J. de Kleer, in *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown, Eds. (Academic Press, New York, 1983), p. 227.
34. T. Winograd, *Commun. ACM* **22**, 391 (1979).
35. M. Sanella, *Interlisp-D Reference Manual* (Xerox Corporation, Palo Alto, CA, 1983).

36. We thank J. S. Brown, J. de Kleer, K. Kahn, G. Kiczales, M. Miller, and J. Shrager for comments on earlier versions of this paper.

Knowledge-Based Systems

RANDALL DAVIS

First developed two decades ago, knowledge-based systems have seen widespread application in recent years. While performance has been a strong focus of attention, building such systems has also expanded our conception of a computer program from a black box providing an answer to something capable of explaining its answers, acquiring new knowledge, and transferring knowledge to students. These abilities derive from distinguishing clearly what the program knows from how that knowledge will be used, making it possible to use the same knowledge in different ways.

WORK IN ARTIFICIAL INTELLIGENCE (AI) HAS OFTEN looked for inspiration to the only easily accessible example of intelligence, human behavior. The earliest attempts to design intelligent programs were heavily influenced by the observation that people seem to make some progress on virtually any task, even those that are unfamiliar. Given problems in symbolic logic or algebra, naïve subjects displayed a consistent set of widely applicable problem-solving methods (*1*). Generality came to be seen as a keystone of human intelligence; intelligence appeared to reside in a small collection of domain-independent problem-solving methods. Programs based on such methods displayed encouraging early success.

It became clear that although these methods provided a useful foundation, they were soon overwhelmed by the complexity of real-world problems. Performance on such problems seemed to require large stores of task-specific knowledge (*2*).

This observation led to a significant shift in emphasis for the part of the field that came to be known as knowledge-based systems, in which work has come to focus on the accumulation, representation, and use of knowledge specific to a particular task. The term knowledge-based is primarily a label for this focus and an indication of the source of the systems' power: task-specific knowledge, rather than the domain-independent methods used in early AI programs (*3*). That knowledge is often incomplete and at times involves inexact judgments, unlike the knowledge that underlies carefully designed algorithms of traditional software. The systems can also be characterized by an architecture and a set of capabilities that result from it, including explanation, knowledge acquisition, and tutoring, as well as problem-solving performance.

Previous discussions have focused largely on performance, describing applications and levels of performance reached (*4*). This discussion considers how these systems have expanded our view of a program, an expansion made possible in part by the ability to use the same knowledge in several different ways. This is demonstrated in the context of a rule-based system, since it is the most familiar technology used for constructing these systems.

Randall Davis is an associate professor at the Sloan School of Management at Massachusetts Institute of Technology and a member of the MIT Artificial Intelligence Laboratory, Cambridge 02139.