Software Reviews

# PC Software for Artificial Intelligence Applications

HELMUT EPP, MARTIN KALIN, DAVID MILLER

Software tools that have been developed in the course of Artificial Intelligence (AI) research are now mature enough to stir interest outside of AI, and relatively cheap but powerful hardware make it feasible to try such tools on a broad range of problems. What AI programmers find essential, all programmers find desirable—fast program development and natural expression of problem-solving knowledge. After contrasting AI and conventional software, we examine three types of AI tool by reviewing representative PC-based commercial products.

AI researchers try to make the computer solve problems that apparently have no direct algorithmic solution. Implementing a particular data analysis algorithm (for example, linear regression) is not an AI problem, but writing a program to select an appropriate data analysis technique is. AI researchers approach a problem by developing a prototype program that is a partial solution. The researchers then modify the prototype to improve its performance. AI researchers have found conventional programming languages, such as BASIC or FORTRAN, deficient with respect to this exploratory style of programming. The deficiency springs both from the types of object and operation supplied in a conventional language and from the tradional programming environment. The objects typically include numbers of various types, numerically encoded characters, arrays, and memory addresses. The supplied operations provide the appropriate manipulations of these objects, for example, adding numbers or testing for equality of characters. These objects and operations are low level in that they mimic what the computer supplies at the hardware level. Finally, the traditional programming environment does not provide the development tools required for fast prototyping, a key part of experimental programming.

Symbolic programming languages, such as LISP, are popular in AI because they improve upon conventional languages by furnishing primitive operations that manipulate symbolic objects and their interrelations. Yet these languages share a deficiency with their conventional counterparts: both are procedural in requiring that the basic steps of problem-solving be encoded in the program by the sequencing of statements. An individual programming statement, such as an assignment of value to a variable, derives its meaning from its position within the sequence, which makes it difficult both to reason about the statement's correctness and to change its meaning. AI programmers try to overcome this deficiency by building languages that allow information about the problem to be expressed as naturally, explicitly, and nonprocedurally as possible. (LISP is so popular in AI partly because it is easily extensible: specialized languages can be built readily on top of it.) A good AI language, besides encouraging the fast prototyping of solutions, should be general enough to cover a broad group of problems.

An AI tool is typically inspired by some problem-solving paradigm. We begin with Personal Consultant Plus (*1*) as an example of a backward-chaining rule system. Next we cover Smalltalk/V (*2*) as an example of an object-oriented system. We conclude with Nexpert Object (*3*) as an example of an integrated or toolkit system. Because the three products address different problem-solving needs, we do not rank them; although we do not compare them against commercial rivals, we regard all three as among the best available.

## Personal Consultant Plus

*Background.* A natural form of expression of problem-solving knowledge is the conditional: IF the situation is like this, THEN this fact is true or this action should be performed. A number of systems have been developed that have the IF-THEN rule or situation-action rule as their major form of representation. We program a solution to a class of problems in one of these rule-based tools by writing a set of rules that collectively describe the problem-solving process. The modular nature of expression in these tools, where, ideally, each rule represents an independent piece of knowledge about the problem, makes it easier to develop programs incrementally.

A given rule can be used in different ways during problem-solving. Consider, for example, a rule that might appear in a program monitoring patient parameters during surgery:

If mean arterial pressure <50 mmHg
and arterial $CO_2$ pressure is low,
Then cerebral blood flow will be reduced.

Used in the forward direction, reasoning proceeds from rule antecedent (the IF part, also called the left-hand side or LHS) toward the consequent (the THEN part, right-hand side or RHS). Suppose that our monitoring system maintains a database of facts. When the antecedent of this rule is discovered to be true (because appropriate values of mean arterial pressure and $CO_2$ pressure have been observed), the rule can be fired, placing the conclusion about cerebral blood flow into the database (where it might cause the antecedent of another rule to be satisfied, a rule, say, whose consequent action would be to update a display and trigger a warning alarm).

A rule can also be used in a backward direction, reasoning from consequent to antecedent. Used this way, the rule given above would be of interest if the monitoring program was attempting to determine the status of cerebral blood flow: the rule's antecedent describes sufficient evidence to conclude that the flow is low. The program then could attempt to determine mean arterial pressure (by direct sensor observation) and whether $CO_2$ pressure is low (a judgment as opposed to a direct measurement). Additional rules would need to be consulted to support such a qualitative conclusion.

Any rule-based programming tool has an inference engine, which is a fixed procedure for manipulating rules. This engine is "programmed" by giving it a set of rules describing the problem. To solve a particular problem, the inference engine decides which rules to apply and when to apply them, based both upon the structure of the set of rules it has been given and the parameters of the particular problem. Thus rules are used opportunistically, as the situation demands, and not in some preordained order.

The various rule-based tools differ primarily in the design of their inference engines. A forward-chaining tool has an inference engine that uses rules only in the forward direction. A backward-chaining tool has an inference engine that uses rules only in the backward direction. Hybrid tools allow each rule to be used in the direction specified by the programmer.

Most backward-chaining tools are specialized to deal with diagnosis problems, in-

Institute for Applied Artificial Intelligence, Department of Computer Science and Information Systems, DePaul University, Chicago, IL 60604.

volving the selection of one or several most likely candidates from a set of alternatives. Such systems share the same basic computational model. Problems are described in terms of parameters, which have values. The antecedents of the rules test the values of parameters, and their consequents make conclusions about the values of other parameters. Problem-solving is goal-driven: some parameter is designated as the overall goal, and the values ultimately concluded for this parameter constitute the diagnosis. The system backward-chains through rules that conclude about the goal parameter, attempting to determine which of those rules have true antecedents. In order to test the antecedent of a rule, the values of the parameters mentioned in it must be known, so these parameters become subgoals that cause additional backward-chaining.

To illustrate the process, consider the MYCIN system for diagnosis of and therapy selection for bacterial infections (*4*). The diagnostic portion of MYCIN is a backward-chaining rule-based system which has a goal of determining which organisms would best explain the patients's symptoms and lab tests. MYCIN has several goal parameters, including **COVERFOR**, whose computed value is a list of organisms which must be covered by the therapy. The rule shown in Fig. 1, in both its internal form and in an English translation, concludes a value for the **COVERFOR** parameter on the basis of values for the **TREATINF**, **EXAMSTAIN**, **SPECSTAIN**, **TYPE**, and **BURNED** parameters. For a conclusion to be made by this rule, the system must determine the values of these antecedent parameters. For some parameters, such as **BURNED**, the value is determined by asking the user directly. Other parameters, such as **TREATINF** and **TYPE**, represent sophisticated conclusions; additional rules must be tried in order to determine their values.

A backward-chaining tool can have its inference engine designed to produce dialogues with the user of an application that seem purposeful and focused. For example, if the inference engine always investigates a parameter such as **COVERFOR** completely before moving on to the next parameter, then all the questions about **COVERFOR** will be asked at one time instead of being interspersed with questions about some other parameter. While the system is using the rule shown above, attention is focused for a while on **TREATINF**, the infection to be treated; focus later shifts to **TYPE**, the type of the organism.

The first major rule-based tool to use backward-chaining was EMYCIN (*4*), developed at Stanford University from the

If: 1) The infection which requires therapy is meningitis,
2) A: A smear of the culture was not examined, or
B: Organisms were not seen on the stain of the culture,
3) The type of the infection is bacterial, and
4) The patient has been seriously burned
Then: There is suggestive evidence (.5) that pseudomonas-aeruginosa is one of the organisms which might be causing the infection.

```
PREMISE: ($AND (SAME CNTXT TREATINF MENINGITIS)
               ($OR (NOTSAME CNTXT EXAMSTAIN)
                    (NOTSAME CNTXT SPECSTAIN))
               (SAME CNTXT TYPE BACTERIAL)
               (SAME CNTXT BURNED))
ACTION:  (CONCLUDE CNTXT COVERFOR PSEUDOMONAS-AERUGINOSA
                   TALLY 500)
```

**Fig. 1.** Example of a rule, both in English translation and in code, from the MYCIN program.

MYCIN system by abstracting away all the specifically medical knowledge and leaving a language for writing rules and the inference engine. Personal Consultant Plus (PC-Plus) is a direct descendant of EMYCIN, that is somewhat better organized than the original, has certain obvious missing pieces supplied, and is adapted to the PC environment.

All control in a pure backward-chaining rule-based system springs from attempting to determine the values of parameters; therefore, the overall expressiveness of the rules is determined by how much control over this process is given to the application programmer. Of particular interest is control of the dialogue; more natural dialogues make the problem-solving process seem more transparent.

*Operation.* In PC-Plus, we control the inference process by setting various parameter properties. For example, the absence of the **PROMPT** property on a parameter means that the user will not be asked to provide the parameter's value. Having a **PROMPT** and an **ASKFIRST** property means that the user will be asked for a value before any rules are tried, whereas a **PROMPT** and no **ASKFIRST** property means the user will be asked for a value only after rules are tried. A **METHOD** property specifies a procedure to call or to calculate a value. A **DEFAULT** property can be specified to provide a value if none of these procedures work. PC-Plus allows control over how questions are asked. Restrictions can be placed on the kinds of values users can type in (symbolic values chosen from some set, arbitrary text typed in by the user, or numeric values within a certain range) and how many values are allowed (**SINGLEVALUED** parameters can only take on one value, **MULTIVALUED** ones can have several values). **PROMPT**s can be constructed automatically by PC-Plus from **TRANSLATION**s provided by the programmer. For example, the **SPECSTAIN** parameter might have the translation "Organisms were seen on the stain of the culture" from which the system would construct the query "Were organisms seen on the stain of the culture?" The programmer can also create graphics (using an outside graphics package) and incorporate them into the application as prompts, help, and display of final results. A recently added Images utility extends the graphics capabilities to allow, for example, input of text by forms and of numeric values by images of dials and thermometers.

In PC-Plus, parameters are grouped into frames (called "contexts" in EMYCIN) allowing relevant parameters to be considered together. Frames are connected into a hierarchy. In the MYCIN system, for example, the **PATIENT** frame at the root of the hierarchy contains parameters related to general patient information, such as **NAME** and **AGE**. Subframes subordinate to **PATIENT** group parameters related to different kinds of cultures (positive, negative, pending, or prior). A frame is a template for an entity, called an instance of the frame, to be created when the application is run. The instance contains the actual parameter values for a given case. Multiple instances of a frame can be created; thus there can be multiple instances of **POSITIVE-CULTURE**, each having its own values for a common set of parameters. Parameters designated as **INITIALDATA** parameters within a frame will be asked about when an instance of that frame is first created. The system will then attempt to find the values of any designated **GOAL** parameters. In MYCIN, **NAME** and **AGE** are **INITIALDATA** for the root **PATIENT** frame and **THERAPY** is the **GOAL**. In the **POSITIVE-CULTURE** frame, **DATE-TAKEN** and **SITE** are **INITIALDATA** and **IDENTITY** is the goal. These choices help structure the dialogue with MYCIN to meet physician expectations; general questions about the patient are asked to begin the diagnosis, and identifying features of a culture are asked whenever the culture is first mentioned.

Frame instantiation, that is, putting pa-

rameters into a frame, is performed automatically by the inference engine; for example, a rule concluding for **THERAPY** might mention **IDENTITY**, which will trigger the creation of an instance of **POSITIVE-CULTURE**, since **IDENTITY** is a parameter associated with that frame. Frame instantiation can be placed under the control of the user (who could be asked "Are there any positive cultures?" or "Are there any more positive cultures?"). More sophisticated behavior can be induced by having rules explicitly force instantiation.

The order in which rules are tried is a major determinant of dialogue focus. If arbitrary ordering of the rules leads to unfocused dialogue, the programmer can rank-order the rules on a numeric scale, or explicitly specify that certain rules are to be tried before others. Moreover, this information can be decided during a consultation, allowing a more promising line of reasoning to be pursued first.

*Discussion.* PC-Plus provides a development interface that helps manage application development. There are windows for editing parameter properties, frame properties, rule definitions, and so forth. Selection of editing commands appropriate to the window is done through a menu or through control keys. The windowing system itself is supplied with PC-Plus and predates current window-mouse–oriented interfaces for PC-DOS. The windows stack on top of each other; only the top-most window can be acted on and it must be exited in order to access lower windows. On a more positive note, the interface does help the application programmer, for example, by performing syntax checks on rules as they are defined, allowing immediate reediting in case of error, and prompting for the definitions of parameters when they are first encountered in rules. Rules are written in ARL (Abbreviated Rule Language). The **TRANSLATION**s for parameters provided by the programmer allow English paraphrases of the rule to be generated so that the programmer can verify the correctness of the ARL statements. (These paraphrases also provide a simple explanation mechanism for the user.)

PC-Plus is written in TI-SCHEME, a dialect of LISP. All LHS tests and RHS actions actually translate into LISP function calls. The application programmer can add new tests and actions by writing LISP code. These extensions can even be given translation templates for use in rule translations. PC-Plus has added a rich set of extensions to the original EMYCIN capabilities, including access to dBASE-III and DOS. PC-Plus sells for $2950. A reduced-functionality version called PC-Easy is available for $495. It provides a nice way for the potential user to experiment with this technology, with the option of upgrading to PC-Plus if desired.

PC-Plus is an expressive, flexible tool for producing diagnostic programs. Its only drawback is not specific to it, but rather is common to all programming tools built on a single paradigm: as long as the solution to the problem reasonably matches the mechanisms provided by the tool, solutions can be coded cleanly and succinctly, but as the match becomes less close, the clarity of the resulting program declines dramatically. In particular, purely backward-chaining rule-based systems have difficulty expressing complex diagnostic strategies. Although PC-Plus, through its ability to control frame instantiation and ordering of rules, is certainly among the best of any of the backward-chaining tools in this capability, its limits come from the paradigm that inspired it.

## Smalltalk/V

*Background.* Traditional programming consists of applying various program steps to data. These steps are combined to form procedures. Grouping procedures together with the data affected by them leads to the concept of an object: an object has a set of operators and a state ("private memory") that can retain the effect of the operations. For example, we can define a rational number as an object with a generic syntax (Fig. 2).

```
A RATIONAL_NUMBER is kind of NUMBER Object with data
        NUMERATOR:     INTEGER;
        DENOMINATOR: INTEGER range 1.. INTEGER'LAST;
  and   with
        operation     "="      (X,Y: RATIONAL_NUMBER) return BOOLEAN;
        operation positive"    (X: RATIONAL_NUMBER) return BOOLEAN;
              Begin
                 IF X.NUMERATOR > 0 return TRUE else return FALSE
              End operation
        operation     "+"      (X,Y: RATIONAL_NUMBER) return RATIONAL_NUMBER;
              Begin
                 return ( new( X.NUMERATOR * Y.DENOMINATOR +
                               X.DENOMINATOR * Y.NUMERATOR,
                               X.DENOMINATOR * Y.DENOMINATOR ) )
              End operation
        operation     "–"      (X,Y: RATIONAL_NUMBER ) return RATIONAL_NUMBER;
        operation     "/"      (X,Y: RATIONAL_NUMBER ) return RATIONAL_NUMBER;
end object definition;
```

**Fig. 2.** Definition of an object with the use of a generic syntax.

For simplicity, we show definitions for only two operations. Although a rational number is just a pair of integers, the discipline of objects requires that neither can be accessed without explicit accessor functions. As our definition stands, no rational numbers may be created because we lack a "create" function:

```
operation "new" (X,Y: INTEGER)
return RATIONAL_NUMBER is
      Begin
         return (X,Y);
      End;
```

The statement

```
X:= rational_number      new (4, 5):
```

creates a rational number and assigns it to **X**. Invoking an object's operations, such as the "+" operation, is called message passing. The same message pattern may invoke different operations depending on the receiver. Operators such as "/" and "+" are overloaded in that they have different meanings for different objects. For example, "+" for rational numbers differs from "+" for real numbers. The ability to send the same message to different object types, and having each respond appropriately, is known as polymorphism.

SIMULA (1966) introduced the notion of classes of objects, their hierarchy, and the concepts of inheritance. A class is a collection of similar objects and may be the specialization of superclasses from which it inherits functionalities and values. For example, **RATIONAL_NUMBER** was defined as a kind of **NUMBER**, which may support various operations that could be inherited by **RATIONAL_NUMBER**. For example, we can define the operation ">" for numbers so that **X>Y** returns true if **X – Y** is positive. The object **RATIONAL_NUMBER**, already furnished with the "–" and positive operations, now can inherit the ">" operation. The inheritance works as follows: If **X** and **Y** are **RATIONAL_NUMBER** objects, then the expression "**X>Y**" passes the ">" message to **X** with parameter **Y**. Because there is no ">" operation explicitly defined for **RATIONAL_NUMBER**, an object-oriented system looks for the definition in the subsuming class. In the class NUMBER, ">" could be defined as "**X – Y positive**," which invokes the messages "–" and **positive** that are explicitly defined for **RATIONAL_NUMBER**. In case an operation is defined at more than

one level, the first encountered definition is applied.

An object-oriented language supports objects, a hierarchy of classes, and inheritance. Object-oriented languages combine both data and computation in single units ("objects") that interact along well-defined interfaces by way of messages. Objects in this sense are also called Actors in a distributed environment. The first uniformly object-oriented systems resulted from two research efforts: Smalltalk from the Learning Research Group at Xerox Parc and the Actor model of computation from the AI laboratory at Massachusetts Institute of Technology. We examine the Smalltalk approach to object-oriented programming with Smalltalk/V.

```
Form subclass: #Light
  instanceVariableNames:
    'location status '
  classVariableNames: ''
  poolDictionaries: ''

Light class methods
create
 ^self new width:20 height:20
Light methods

isOn
 ^status

setOn
    status:=true.

placeAt: aPoint
  location:=aPoint

isOff
 ^status not

setOff
    status:=false

display
  self displayAt:location

turnOff
  self isOn ifTrue:[self setOff;reverse;display]

turnOn
  "if a light is turned on then the changed message will invoke the update: message
in all dependent lights"
  self isOff ifTrue:[self setOn; reverse;changed;display]

update: aLight
Light == self ifFalse: [self turnOff]

Object variableSubclass: #TrafficLight
  instanceVariableNames:
    'position lights '
  classVariableNames: ''
  poolDictionaries: ''

TrafficLight methods

lights: numberOfLights at: place
"this method sets the instance variables of TrafficLight and displays it at place"
lights:= Array new:(numberOfLights max: 1).
        lights at: 1 put:(( Light create)setOn;placeAt: place ;display).
"the other lights are displayed below the first and set off"
2 to: numberOfLights do:
  [:index|lights at: index put: ((Light create)setOff;reverse;
    placeAt:( place + ( 0 @ 30 * (index - 1))); display)].
"iterating over the array lights we create dependencies between each light
    and all the others .~~ means not equal"
  lights do:
    [:eachlight | lights do:
        [:dependentlight | eachLight ~~dependentLight
    ifTrue:[eachLight addDependent: dependentLight]]]

turnOn: lightNumber
  (lights at: lightNumber) turnOn.
```

**Fig. 3.** Definition of a class **Light** and then the class **TrafficLight** with Smalltalk/V.

*Operation.* Virtually every component in a Smalltalk system is an object. The windowing facilities, editors, graphics displays, compiler, class definitions, and basic data types are all objects. For example, **number** is an object that can receive messages. The expression

```
3 + 5
```

sends the message + to the object **3** with parameter **5**. The expression

```
1 arcTan * 4
```

sends the message **arcTan** to the object **1** and then the message * to the result with parameter **4**, which yields **3.14159**. Successive messages can be sent in left to right order so that

```
3 + 4 * 5
```

yields the **number** object **35**, not **23**.

Smalltalk has more than 100 built-in classes, all of which are subclasses of the class **Object**. For example, the built-in class **Point** is defined by:

```
Object subclass: #Point
instanceVariableNames:
'x y'
classVariableNames''
poolDictionaries:''
```

This definition states that **Point** is a subclass of **Object**. Every instance of **Point** gets private data variables **x** and **y**. There are no class-wide variables to be shared by all the instances (**classVariableNames** is empty) and no variables to be shared with other classes (**poolDictionaries** is empty).

In Smalltalk we invoke an operation by sending a message. Message passing is, in effect, procedure invocation. A message's content, analogous to a procedure's body, is a **method**. The class **Point** has several methods, which define accessor operations **x** and **y** as well as a create operation **@**. The expression

```
Dot:= 3 @ 5
```

creates a point and assigns it to a global variable **Dot**. The expression

```
Dot x
```

returns the object **3**.
The method "*" is defined by:

```
* scale
    "Multiply a point by scale:
    if scale is a Point,
    then do component-wise multiplication
    else do scalar multiplication."
    scale class == Point
      ifTrue: [ ^(x * scale x) @ (y * scale Y)]
      ifFalse: [ ^(x * scale) @ (y * scale)]
```

The parameter **scale** is interogated by the message **class** as to whether it is a point or a scalar; **ifTrue** and **ifFalse** are conditionals,

each followed by expressions to be evaluated; the ^ causes a value to be returned. We can quickly redefine operations or add them. For example, we could redefine "*" for points, but this may not be wise, because the system may use the original. When we add or redefine methods, they are compiled and installed into the method dictionary.

We now present a longer example to give the flavor of Smalltalk programming, adapted from (*15*). We simulate a traffic light in which only one light is turned on at a time. We use some preliminary objects. The built-in class **Form** represents a two-dimensional array of bits. The **width: height:** message creates an instance of Form. Evaluating the expression

```
Panel1: = Form width: 100 height: 200
```

and performing the operation:

```
Panel1 displayAt: 0 @ 0
```

displays a white rectangle in the upper left-hand corner of the screen. Semicolons are used to send multiple messages to the same object. If we evaluate

```
Panel1 reverse: displayAt: 0 @ 0
```

the white rectangle will be replaced by a black one.

We define a class **Light** and then the class **TrafficLight** in Fig. 3.

If we evaluate the expressions

```
Tr: = TrafficLight with:3 at: 100 @ 100.
Tr turnOn: 2.
```

we first get a vertical array of three squares with the top square black and the others white. The second expression renders the second square black and the others white.

*Discussion.* What cannot be easily described about Smalltalk/V is the combination of screen-oriented, mouse-driven tools such as editors, browsers, inspectors, and such that make the process of programming and debugging easy. Most of Smalltalk is written in Smalltalk and can be modified by the user. Potential crashes resulting from experimentation are reversible by using the CHANGE.LOG file that records changes to the system. The collection of all the objects form the system IMAGE and can be saved from a session in order to retain changes. As shipped, Smalltalk/V comes with about 7,000 objects and allows about 30,000 more. Although this seems like a large number, it becomes insufficient for serious applications. A forthcoming version takes advantage of protected memory on AT-class machines and thus can support a much larger number of objects. The new version also has color as a standard instead of an option. The part of Smalltalk not written in Smalltalk is a virtual machine that interprets so-called "primitive" methods represented by byte codes. Not all of the 256 possible byte codes are used by the virtual machine, however, and the remainder are available for user-written assembly language routines. Large programs tend to run slowly in Smalltalk because of the underlying interpreter and the use of objects to implement even low-level mechanisms.

For IBM-compatible PCs, Xerox Corporation no longer licenses any commercially available implementation of Smalltalk-80, which makes Smalltalk/V the de facto standard. Implementations of Smalltalk-80 are available for the MacIntosh. The two languages differ slightly so that books and manuals on Smalltalk-80 (*6, 7*) can be used in conjunction with the helpful but limited Smalltalk/V documentation. The standard Smalltalk/V package does not come with the multiprocessing primitive operations of Smalltalk-80 that are often used for discrete event simulation, but some are included in an extension kit, which also includes a Prolog and a rule-based shell. Smalltalk has a small syntax and the language as a whole, including the use of message passing to invoke procedures, is easily learned. The challenge to the programmer is to master the extensive built-in classes that are the analogs of libraries that come with other "small" languages.

We believe that future programming systems will absorb the object-oriented paradigm, a move that is already underway. There are object-oriented extensions for Pascal, C, and LISP. (Microsoft, for instance, has hinted that it is working on an object-oriented BASIC!) Effective object-oriented programming requires the same work as any other type—proper decomposition and representation of the problem. The localization of data and procedures is attractive, because it simplifies procedure invocation to message passing, and inheritance enables rapid changes to object representation. Smalltalk/V costs only about $100 and the extension kit is about another $50. At these prices, it is an especially attractive object-oriented system.

## Nexpert Object

*Background.* A single-paradigm tool might be suitable for one part of a problem, but not for all of it; a second tool would be needed for another part, and so on. An AI toolkit is meant for problems that seem to need a mix of specialized tools. Minimally, a toolkit should have a hybrid rule system, object-oriented programming, and access to a general-purpose language. Additionally, a toolkit should interface with conventional tools such as databases, spreadsheets, graphics packages, and word processors. Nexpert brings a state-of-the-art toolkit to the PC at a reasonable price. (Although available on workstations and larger machines, Nexpert is reviewed here as a PC-based product.) We review its component tools individually, but with an eye on the integration issue.

*Operation.* Nexpert's main tools are an object-oriented system and a hybrid rule system. Its object-oriented component derives from a generic AI tool known as a frame-based system. Frames can represent arbitrary entities and collections of entities (for example, Isaac Newton, mass, or the set of natural numbers). A frame has an arbitrary number of slots, each with a value. The frame representing Isaac Newton might have a slot called **profession** with **alchemist** as its value, whereas the frame representing the set of natural numbers might have a slot called **smallest_member** with 1 as its value. A slot may have a procedure as its value. This is known as procedural attachment. For instance, a **natural_numbers** frame might have slot called **successor_generator** whose value is a function that expects a natural number **n** and outputs **n + 1**. Frames can occur in an inheritance network that propagates properties and values. For example, any member of the set **natural_numbers** could inherit both the **successor_generator** slot and the function that is this slot's value.

The jargon of frames is helpful because different object-oriented systems (for example, Smalltalk/V and Nexpert) may use either the same term (such as "object") to mean different things or different terms (such as "instance variable" and "property") to mean the same thing. Nexpert has two types of frames: classes, which are collections, and objects, which typically belong to one or more classes. In Nexpert, slots are called "properties." (In Smalltalk/V, by contrast, a class slot is called a "class variable" and an object slot is called an "instance variable.") A Nexpert class or object can have arbitrarily many properties, and the inheritance network of objects and classes can be arbitrarily complex. Nexpert supports multiple inheritance, namely, an object or class can inherit properties and values from more than one source. For example, suppose there is class **Transport_Tasks** that has a **minimum_duration** property with a value of 18 (minutes), another class **Single_Resource_Tasks** that has a **priority** property with a value of **high**, and an object **Transport_Task27** that is a member of both classes. The object could inherit the **minimum_duration** property and value from **Transport_Tasks** and the **priori-**

**ty** property and value from **Single_Resource_Tasks**. (It is also possible to inherit a property without the value.) A class can propagate properties and values down to its member objects, and objects can propagate properties and values up to classes that subsume them; inheritance thus takes the form of either specialization (class to object propagation) or generalization (object to class propagation).

Every property of an object or class has a value, which defaults to **Unknown**. A property also has properties of its own ("meta-slots") that determine its behavior in inheritance, initialization, and modification. For example, suppose the class **Transport_Tasks** should have its **minimum_duration** property initialized to 12 instead of **Unknown**. On this property we set the meta-slot called **Order of Sources** to **Init-Value** and specify 12 as the initial value, which then replaces **Unknown**: also, we can specify that this value be inherited down to all subclasses and members. Meta-slots also can be used to implement a demon, which is a piece of code that monitors a property and reacts appropriately to changes in its value. For instance, suppose that a warning would be appropriate whenever a task's duration exceeds some threshold and that a function has been written (for example, in FORTRAN or C) to compute thresholds for different tasks. Through the **If Change** meta-slot on a task's **current_duration** property, we can have Nexpert invoke the function whenever this property's value is updated. We could pass the function whatever data seem appropriate for its computations or database lookups. The function, in turn, could issue a warning, return a value, or even create a new Nexpert object. Demons are a type of procedural attachment that can take us outside Nexpert into a conventional language such as C or FORTRAN.

Rules are Nexpert's preferred way to process objects. In Nexpert, a rule may be used either forward or backward. It has the form:

```
IF      condition1 AND condition2
        AND . . . conditionN
THEN    conclusion AND
        action1 AND action2
        AND . . . actionN
```

(The actions are optional.) The rule below assume a class **Machines** and an object **Shutdown_Operation**:

```
If    Shutdown_Operation.status  is
      idle  And
      ⟨Machines⟩.status is broken
Then  schedule_a_shutdown  is
      confirmed  And
      Shutdown_Operation.status  is
      set to busy  And
      Shutdown_Operation.target  is
      set to ⟨Machines⟩.ID  And
      ⟨Machines⟩.status is set to under-
      repair  And
      ⟨Machines⟩.processing_state  is
      set to idle  And
      Execute predict-downtime@
      atomid=⟨Machines⟩.Type
```

A left-hand side (LHS) condition typically tests the value of a property for some object or class. In the sample rule, the first condition tests whether the **status** property of object **Shutdown_Operation** has **idle** as its value. The second condition screens for any object in **Machines** whose **status** property has **broken** as its value. Angle brackets designate a quantifier. For example, the condition

```
⟨Machines⟩.status  is broken
```

tests whether at least one object in class **Machines** has a value of broken for its **status** property. Curly brackets designate another type of quantifier. For instance, if we wanted to check whether every object in class **Machines** had a value of **broken** for its **status** property, we would see this pattern

```
{Machines}.status  is broken
```

instead of the earlier one. Quantifiers can be mixed, allowing us to draw distinctions in the LHS such as these:

1) that at least one machine whose status is broken have an expected repair time shorter than 3 hours
2) that every machine whose status is broken have an expected repair time shorter than 3 hours

Mixed quantification lends great expressive power to LHS conditions.

A rule normally fires in a forward direction whenever all its LHS conditions are satisfied, although Nexpert allows the user to control forward-chaining so that, for example, a rule fires precisely if its conditions do not hold. An LHS condition may invoke an external procedure and pass arguments to it:

```
If    there is evidence of ⟨drug⟩.FDA_ap-
      proval  And
      ⟨drug⟩.Toxicity  is Unknown  And
      Execute get-drug-toxicity@atomid=
      ⟨drug⟩. Toxicity,
      ⟨drug⟩.ID_number,     And
      ⟨drug⟩.Toxicity  is low
Then
      good_medicine is confirmed  And
      ⟨drug⟩.Final_rating  is set to approved
```

The first two LHS conditions screen for drugs with FDA approval but unknown toxicity, whereas the third invokes a function (written outside Nexpert) and passes it the drug's **Toxicity** and **ID_number** properties. We assume the function **get-drug-toxicity** returns a value such as **low** that becomes the value of the **Toxicity** property. The third LHS condition then checks whether the drug has low toxicity. The external procedure thereby serves two purposes: it imports data from outside Nexpert and acts as a user-defined test. The external procedure could do other work as well; for example, it might print a list of competing drugs with the same toxicity.

A rule's RHS has a conclusion ("hypothesis" in Nexpert jargon) and, optionally, actions. The hypothesis defines the rule's major topic and, as such, can be used to control backward and forward chaining. If two rules share the same RHS hypothesis, then both become candidates in a backward chain on that hypothesis. An LHS condition in one rule also can be the RHS hypothesis in another. Consider the rules below that, for emphasis, have a simplified syntax. They illustrate how rules can be linked, through RHS hypotheses and LHS onditions, for a mix of backward and forward chaining.

```
R1: If big_increase_in_communication
      _traffic     and
      increased_telemetry_testing
      Then ELINT_evidence
R2: If infrared_signals_exceed_threshold
      and
      no_sunspot_interference
      Then space_sensor_detection
R3: If space_sensor_detection      and
      ELINT_evidence
      Then likely_missile_attack
R4: If likely_missile_attack
      Then launch_seabased_ICBMs
R5: If likely_missile_attack
      Then launch_landbased_ICBMs
```

We can invoke backward chaining on R3 by making likely_missile_attack a goal to be confirmed. The backward chaining invokes R1 to confirm ELINT_evidence and R2 to confirm space_sensor_detection. If the LHS conditions of R1 and R2 are satisfied, then the goal likely_missile_attack is confirmed through backward chaining and hence can be used for forward chaining in R4 and R5. Rules such as these that share conditions or hypotheses have what Nexpert calls strong links. There are weak links as well, which let us declare that rules are related although they do not share hypotheses or conditions. To build weak links, we list for a hypothesis H all of the other hypotheses that Nexpert is to consider when processing H; for example, whenever it backward chains with H as a goal.

A rule's RHS actions can create and delete objects, set an object's property to a value,

display graphics, alter an inheritance strategy, prompt the user, load a knowledge base, execute an external procedure, and so on. In our earlier example, the RHS actions are

**Shutdown_Operation.Status is set to busy**
**Shutdown_Operation.Target is set to ⟨Machines⟩.ID**
**⟨Machines⟩.Status is set to under-repair**
**⟨Machines⟩.Processing_State is set to idle**
**Execute predict-downtime@atomid=⟨Machines⟩.Type**

The first four actions set an object's property to some value, and the last invokes the external procedure **predict-downtime**. The syntax for an RHS procedure call is the same as for an LHS one.

An RHS action can be used to control rule processing as an application runs. For example, consider the case in which two rules share an RHS hypothesis:

R1: LHS1 → H and Actions1
R2: LHS2 → H and Actions2

Suppose that, if R1 fires, we want any subsequent backward chaining to be exhaustive, namely, we want the system to backward chain on every rule that has the goal hypothesis, regardless of whether the backward chaining succeeds or fails on a given rule. If R2 fires, however, we want to backward chain only until we succeed once; at that point, the backward chaining halts. Actions1 and Actions2 would implement the different strategies with a Strategy action. Nexpert allows similar dynamic or run-time control over inheritance. Such control adds to problem-solving flexibility.

*Discussion.* A toolkit is meant for problems that seem beyond a single-paradigm tool. Because a toolkit's strength is diversity, its architecture should be open so that we can blend the built-in tools with others. Nexpert is an open system. External procedures can be invoked from a rule's LHS or RHS as well as through the meta-slot mechanism; Nexpert furnishes procedures to access standard database systems such as dBASE III PLUS. Nexpert itself can be called as a procedure with the calling program enjoying full control over the tools and the environment as a whole. The natural way to use an AI toolkit is as a system integrator: it probably makes more sense to call out of Nexpert than to call it from the outside, but it is important that both are possible. Nexpert documentation includes source code for the functions that implement its open architecture. Nexpert has a modern development envrionment. Its menu-bars and pull-down menus list choices in seven basic categories (for example, Edit and Report), and its pop-up windows provide appropriate areas in which to edit rules, display inheritance hierarchies, inspect an object or class, alter the environment, run or debug an application, and so on. On the Apple MacIntosh, Nexpert uses the host machine's graphical interface; on IBM-type machines, it requires Microsoft Windows and the vendor recommends a RAM disk. Nexpert is written in C, which makes it relatively easy to port the product to different computer systems; in fact, Nexpert is a library of C modules that implement the built-in tools and provide external interfaces. Nexpert applications are portable across the machines on which it runs; an application developed on, say, an IBM PC can run on a Sun workstation and vice-versa. At a cost of $5000, the PC version is not cheap, but Nexpert can compete with the most advanced AI toolkits that not only cost significantly more but also require more expensive workstation or mainframe environments. As a PC-based toolkit, Nexpert has exceptional features: object-oriented programming with rich pattern-matching capabilities; multiple and bi-directional inheritance; run-time control over inheritance, meta-slots, and procedural attachment; rule-based programming with backward and forward chaining; declarative and dynamic control over rule processing; LHS and RHS external procedure calls; full integration of built-in tools; a modern development environment; and an open architecture.

## Summary

Our review has emphasized that AI tools are programming languages inspired by some problem-solving paradigm. We want to underscore their status as programming languages; even if an AI tool seems to fit a problem perfectly, its proficient use still requires the training and practice associated with any programming language. The programming manuals for PC-Plus, Smalltalk/V, and Nexpert Object are all tutorial in nature, and the corresponding software packages come with sample applications. We find the manuals to be uniformly good introductions that try to anticipate the problems of a user who is new to the technology. All three vendors offer free technical support by telephone to licensed users.

AI tools are sometimes oversold as a way to make programming easy or to avoid it altogether. The truth is that AI tools demand programming—but programming that allows you to concentrate on the essentials of the problem. If we had to implement a diagnostic system, we would look first to a product such as PC-Plus rather than BASIC or C, because PC-Plus is designed specifically for such a problem, whereas these conventional languages are not. If we had to implement a system that required graphical interfaces and could benefit from inheritance, we would look first to an object-oriented system such as Smalltalk/V that provides built-in mechanisms for both. If we had to implement an expert system that called for some mix of AI and conventional techniques, we would look first to a product such as Nexpert Object that integrates various problem-solving technologies. Finally, we might use FORTRAN if we were concerned primarily with programming a well-defined numerical algorithm. AI tools are a valuable complement to traditional languages.

### REFERENCES AND NOTES


1. Personal Consultant Plus, Texas Instruments, Data Systems Group, P.O. Box 2909, M/s 2195, Austin, TX 78769-2909.
2. Smalltalk/V, Digitalk, Inc., 5200 West Boulevard, Century Blvd., Los Angeles, CA 90045.
3. Nexpert, Neuron Data, Inc., 444 High Street, Palo Alto, CA 94301.
4. B. G. Buchanan and E. H. Shortliffe, Eds. *Rule-Based Expert Systems* (Addison-Wesley, Reading, MA, 1984).
5. T. Kaehler and D. Patterson, *A Taste of Smalltalk* (Norton, New York, 1986).
6. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation* (Addison-Wesley, Reading, MA, 1983).
7. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment* (Addison-Wesley, Reading, MA, 1983).