AI Magazine Volume 5 Number 3 (1984) (© AAAI)

Applications Development Using a Hybrid AI Development System

John C. Kunz
Thomas P. Kehler
Michael D. Williams

IntelliCorp
707 Laurel Street
Menlo Park, CA 94025

Abstract

This article describes our initial experience with building applications programs in a hybrid AI tool environment. Traditional AI systems developments have emphasized a single methodology, such as frames, rules, or logic programming, as a methodology that is natural, efficient, and uniform The applications we have developed suggest that naturalness, efficiency and flexibility are all increased by trading uniformity for the power that is provided by a small set of appropriate programming and representation tools The tools we use are based on five major AI methodologies: frame-based knowledge representation with inheritance, rule-based reasoning, LISP, interactive graphics, and active values Object-oriented computing provides a principle for unifying these different methodologies within a single system.

As a result of our applications development experiences, we are beginning to use a development methodology that emphasizes early prototype development, incremental refinement of the problem description, use of multiple integrated solution methods, and emphasis on visibility of both the problem-solution process and the explicit description of the problem domain. The benefits of using this hybrid development methodology include natural and explicit knowledge representations, flexible user-system interaction, and powerful explanation facilities through use of interactive graphics. We present an example to motivate our discussion.

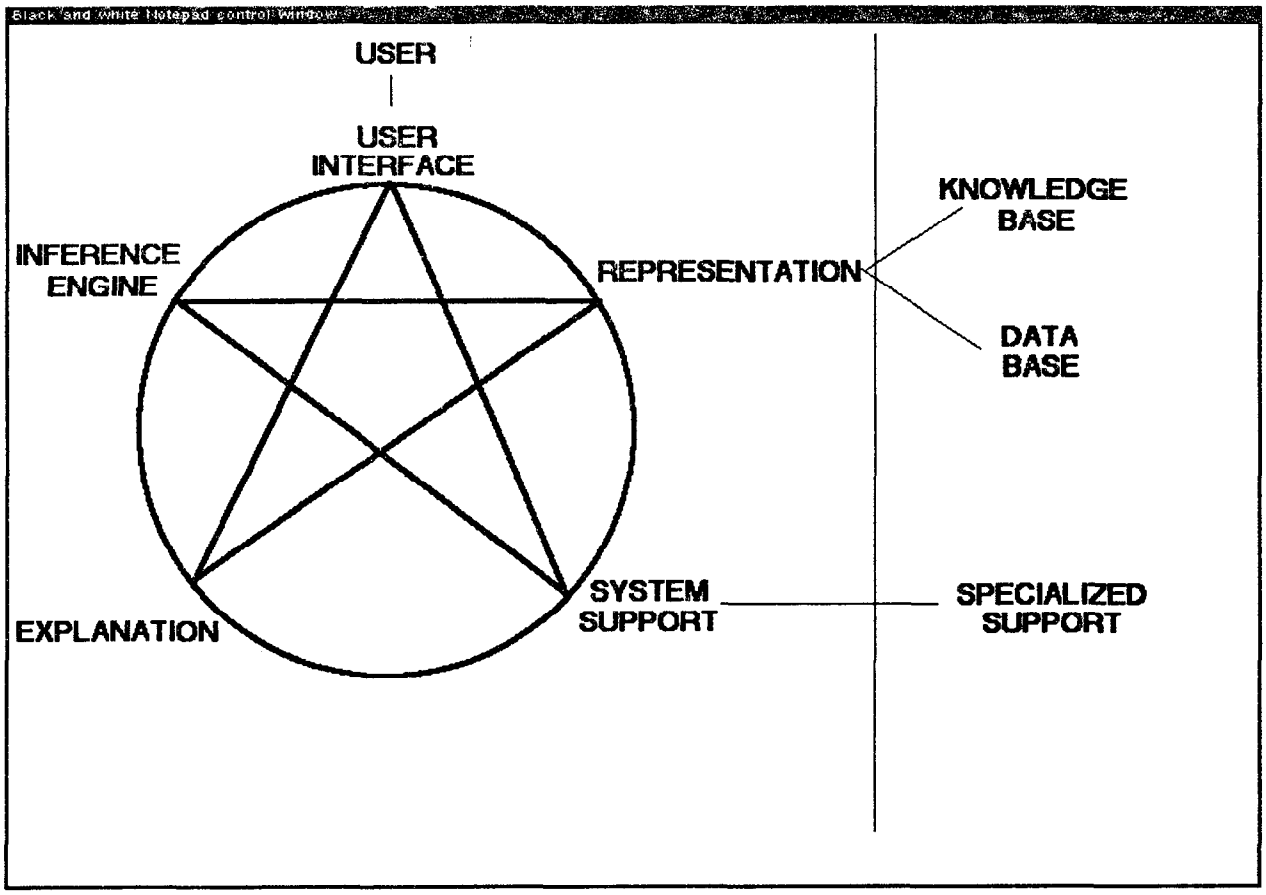
Argument for Hybridization

Workers in AI often express a strong preference for one programming methodology over all others, such as rules (*e.g.*, within an EMYCIN framework), or logic (*e.g.*, PROLOG) or functional (*e.g.*, LISP of one one dialect or another), or object-oriented (*e.g.*, SMALLTALK). These methodologies may or may not be appropriate for solving particular representation and reasoning problems.

The argument to support one programming methodology over another usually falls into one or more of three basic categories:

Naturalness of expression. It usually argued that the methodology proposed is a more natural way of expressing some important class of problems. The class seems to vary from one methodology to another, though each class is argued to be of central importance. Often conciseness is the preferred metric for describing naturalness, although it is sometimes argued to have deeper psychological significance or acceptance over a broader class of users (*e.g.*, production rules). These arguments come down to a user interface issue. The user may find it more "natural" or succinct to express or view the content of a system when knowledge is presented in one form or another.

- *Efficiency* One programming methodology is sometimes argued to be more efficient computationally than one of its rivals. Efficiency seems to depend upon the class of problem chosen.



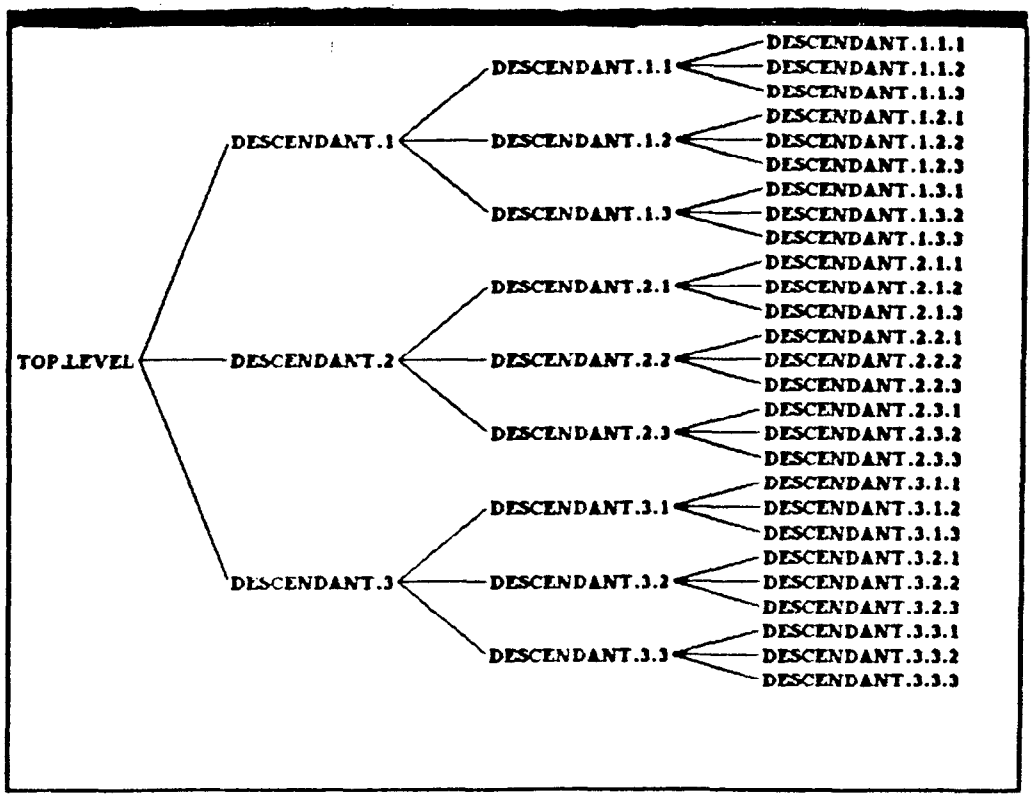
Parts of an expert system development environment (on the left) and of an expert system (on the right).

Figure 1.

- *Uniformity* Since the programming methodology is capable of solving all programs (*i.e.*, is Turing machine equivalent), one methodology should be considered above all others so that users must cope with the complexity of only a single representation and reasoning scheme. This argument is sometimes posed in the form of good scientific practice, parsimony of the underlying representational scheme, and small size of the development environment.

Programming methodologies provide potential solutions to a design problem. Consider the generic problem faced by a domain expert who wants to build a knowledge system: What functions must the designer include in an application system? Figure 1 illustrates our generic view of the parts of a knowledge system application and of the AI development system on which it is based. On the right is the application system itself, including a knowledge base, data about the particular problem, and the set of specialized support routines that every expert system requires. The knowledge base might include some description of the structure and function of a domain and a number of heuristics that describe the way that problems are analyzed within the domain. Specialized support routines might include particular programming functions to define operational meaning of terms such as increased or to provide access to some specialized data structures of an underlying LISP system. The domain expert must supervise the construction of the expert system.

The left side of figure 1 shows the generic parts of an expert system building environment. The inference engine is a widely recognized part of any such environment. In addition, AI application development environments always have a representation component, either explicitly as in a frame-based system, or implicitly as in rule or logic-based systems when the environment commits the user to using rules or logical assertions as a language for representing domain knowledge. The user-interface is a crucial, if sometimes underdeveloped, part of the environment, providing the facilities through which the user enters domain knowledge, asks and answers questions, and views results. The explanation facility shows the user the results of its decision-making process, and ideally it can explain both the structure of the knowledge in the system and the behavior of the system while it operates. Finally, an AI applications development environment provides a number of relatively generic utilities for managing data, interacting with the underlying computer system and the display, and possibly debugging support. Ideally, the domain



An example knowledge base.

Figure 2.

expert need not create a development environment before implementing an AI application system.

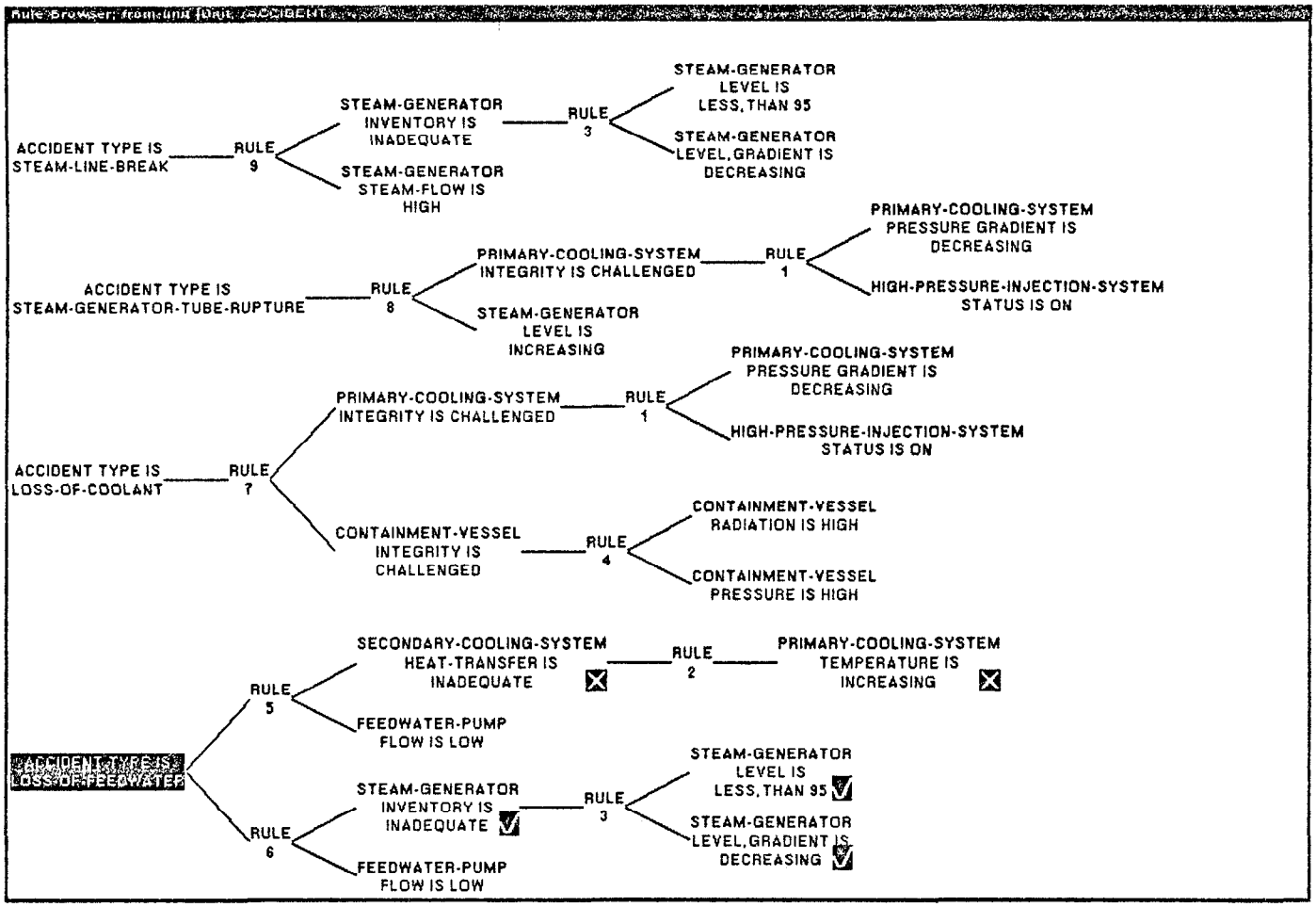
The focus on the AI application system suggests that naturalness and efficiency can be maximized by selecting the most appropriate representation and reasoning technique for the function to be performed.

Sacrificed uniformity is the price to pay for using appropriate methodologies to obtain improved naturalness and representation. While diverse specialized support utilities, such as editors and explanation systems, can be built to support a uniform methodology, the flexibility of a uniform methodology is limited whenever a single representational form becomes opaque, complex or convoluted in performing a particular task. Generally, there are tricks for getting around the limitations of each of the representation and programming methodologies. In contrast, a modest number of complementary methodologies can be supportive of each other and thereby more natural, efficient and flexible than any single methodology. Thus, the designer must trade the effort to support and teach users to use the tricks with the effort to support and teach users to use several methodologies.

The premise of hybrid tool environments is that uniformity of representation and programming methodology can be sacrificed safely to improve naturalness of expression, efficiency and flexibility. Consider the integration of a frame-based representation with a rule system. Imagine a modest system capturing some user's expertise with fault diagnosis of a mechanical system, say a subsystem of an automobile engine or a nuclear power plant. In a hybrid system, an application might include a knowledge base with three levels of hierarchy, as in Figure 2. Assume a branching factor of 3, resulting in a total of 40 concepts $(1 + 3 + 9 + 27)$. Each of the 40 objects might have 3 attributes per concept. In addition, the rule-based part of such a system might include 75 heuristic decision rules, independent of the concept hierarchy. These rules could be distributed throughout the hierarchy, attached to the objects that they describe.

In a pure rule-based system, the concept hierarchy as well as the object attribute values could have been expressed as a set of rules of the form (if (?x = clyde) then (?x isa elephant)), and (if (?y = elephant) then (?y has COLOR gray)). We can calculate the number of rules necessary to describe this structure: 39 to describe the concept hierarchy, 120 to describe the values of attributes (assuming a meta-rule or a rule interpreter that takes special account of inheritance), 60 rules to account for attribute value pair modification (assuming a 50% modification rate), and 204 additional rules if the interpreter does not take special account of inheritance). The sum is 200 to 400 rules, depending on how inheritance is handled, to describe a typical small taxonomy. Finally, as in the hybrid case above, the rule-based knowledge might include 75 heuristic decision rules, independent of the concept hierarchy.

Several elements are notable in this example. First,



Rule graph showing the set of rules being analyzed. Top level goals are on the left, and terminal conditions reported by the automated data acquisition system or the user are on the right.

Figure 3.

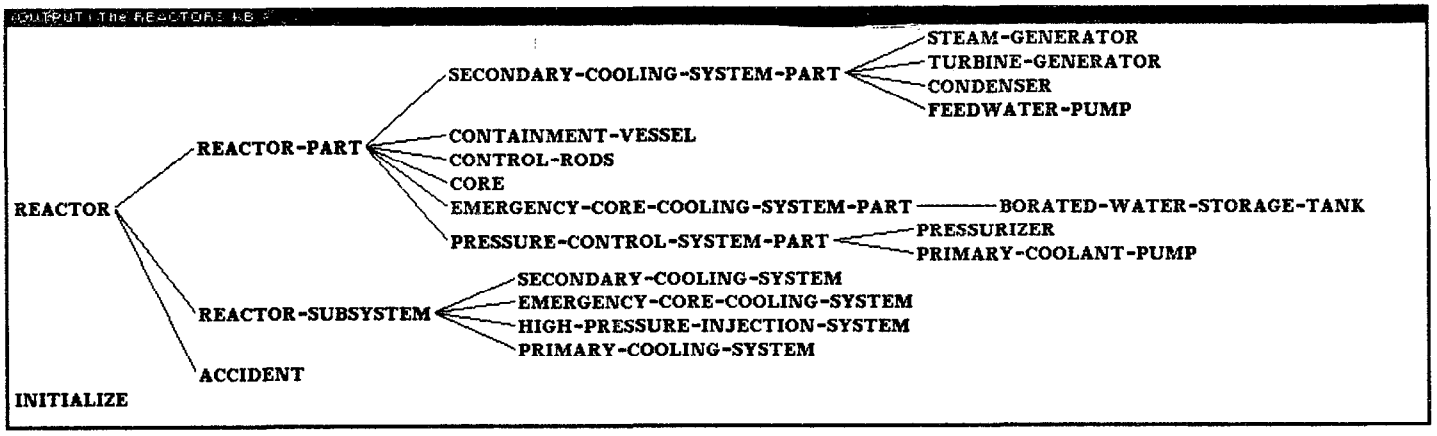
some of the rules defining the domain hierarchy might not be strictly necessary to carry out the decision-making problem, and they might be left out of the knowledge base. The knowledge base then becomes much more opaque to users, however, since the description of the domain has largely been removed. Next, note that 70% to 85% of the total rules in the rules-only version of this structure (200-400 out of 275-475) are used to describe the taxonomy. The heuristic decision rules become overwhelmed by the rules whose principal purpose is definition. In addition, consider the implications on performance of the system dealing with 400+ rules versus 75. In an important sense, inheritance within a frame-based system is a specialized kind of rule, and the implementation of that kind of rule is built into the representation system. Chandrasekaran has argued that a number of such rule/interpreter pairings exist in a society of specialists ranging over both heuristic and descriptive knowledge (Chandrasekaran, 1983).

Similar arguments can be made comparing most of the major programming methodologies and representational techniques available in AI. Each technique has circumstances that make it the most natural form of representation, and for which efficient interpreters can be built.

Given a hybrid tool environment, the problem still remains to understand how to recognize when a particular tool is appropriate and how to apply it.

An Example

Starting with ill-defined problem specifications, AI technology provides a means of developing increasingly precise specifications and implementations of behavioral models for a broad set of decision-making and problem-solving situations. This section discusses an example system to illustrate the way that hybrid AI methods can be used to represent and manipulate a complicated problem in ways that are explicit and powerful. Subsequent sections discuss the specific features and benefits of the knowledge-based system devel-



Symbolic description of the parts of a reactor, as described in the REACTORS KB.

Figure 4.

opment tools that were used in this implementation.

Consider the problem of using a knowledge-based system to analyze and describe the behavior of a complicated system, such as a nuclear reactor. A simple demonstration knowledge base (KB), named REACTORS, analyzes the plant behavior, characterizes its operating mode, and reports when unusual events have occurred.[1]

This article focuses on processing of alarms within a complicated application problem area, such as the REACTORS application, but it does not discuss analysis of plant operation in general.

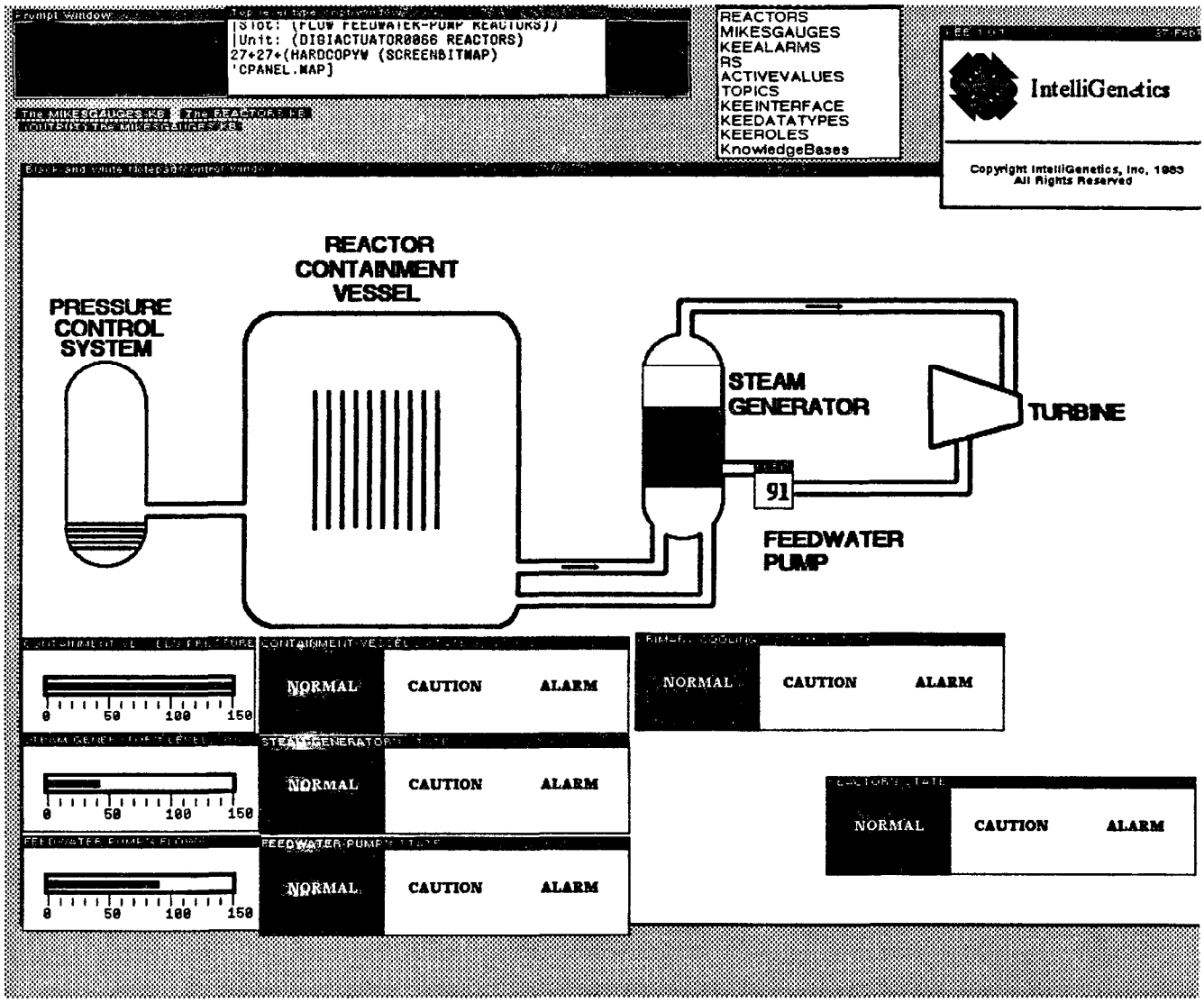
Extending the Technology Base

The rules shown in Figure 3 represent a small subset of the simplified knowledge of an expert in reactor operation and thereby illustrate some of the characteristics of simple rule-based systems. The rules can explicitly represent the context in which they are applicable, and this explicit sensitivity to context is one of the major sources of their power. Their style is very natural for representing heuristic knowledge about the behavior of a system. The active rule graph, shown in Figure 3, explains the invocation of rules to the user. This example of backward chaining has top-level goals to be tested on the left side, such as "Accident type is loss-of-feedwater." Rules that make particular conclusions are shown to the right of the individual goal states, and terminal nodes are on the right side of the rule graph. The darkened box indicates the current focus of attention of the backward-chaining rule system. Check marks note premises that have been found to be true, and Xs denote premises that have been found to be false. New boxes are darkened and checks and Xs are added dynamically as the backward-chainer moves through the rule tree.

[1]The examples of this paper were implemented in KEE, the Knowledge Engineering Environment. KEE is a product of IntelliGenetics.

Limitations of Propositional Rules. The rules of Figure 3 reason with a superficial representation of decision-making behavior of an expert, not with an explicit description of the structure and function of the domain. While similar rules might be written to reason with such structural knowledge, the consideration of such rules suggests one of the major limitations of pure rule-based reasoning: It does not describe structure in a very natural way. More abstractly, the rules describe relations between individual propositions. While propositional representations are powerful, they focus on relations among attributes of objects, rather than on objects themselves, and they do not represent temporal or spatial relations in a natural way. The propositional representation limits the clarity of explanation of the structure and behavior of the model. For example, since the rule-based representation does not suggest easy ways for the user to identify all of the attributes of some object such as a STEAM-GENERATOR, a program designer does not have an obvious approach to deciding how to change a reactor model to accommodate a changed device.

Frame-based representations have been used to describe hierarchical relations such as those found in descriptions of the structure of domains. Frames, or schemas, are very useful for focusing on representation of objects, and they are useful for representing associations of features of objects (Anderson, 1979). The reactor has a number of parts and subsystems, each of which has individual structure and behavior. Figure 4 shows a symbolic description of the parts of the REACTOR'S knowledge base (KB). Lines in Figure 4 connect names of objects that describe classes, such as REACTOR-PARTS, to names of objects that describe a subclass, such as STEAM-GENERATOR. Objects have attributes, and attributes are passed by "inheritance" from class to subclass. Part of the information about any concept lies in its relations to other concepts. Rules provide one way of showing such relations, but the relations between frames shown in Figure 4 provide additional useful information about relations among objects, such as names of the classes from which a particular object



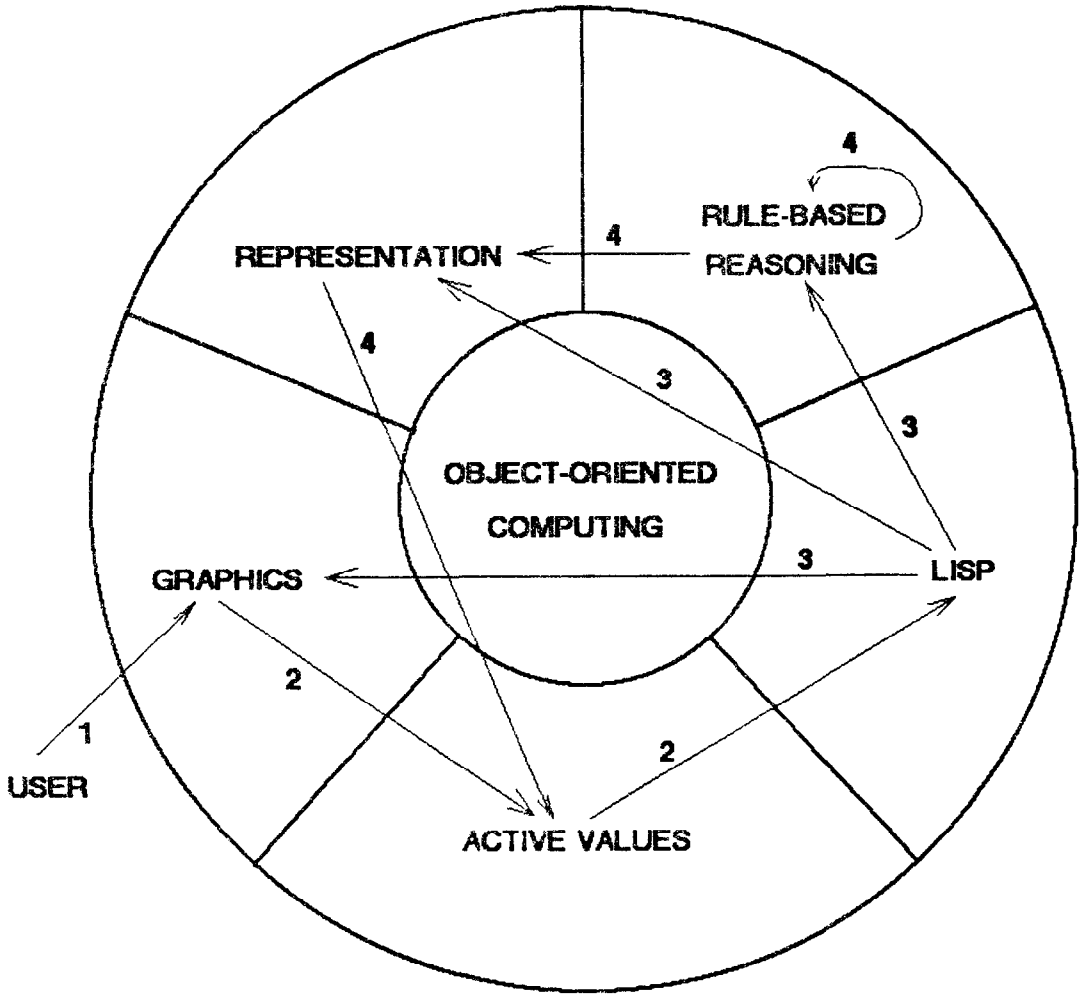
A control panel for the reactor system. Each gauge both reports the parameter value and can be modified by the user with a mouse.

Figure 5.

inherits its attributes and the names of any subclasses that inherit its attributes. In the case of updating a reactor model to include a new steam generator, the user would simply describe the STEAM-GENERATOR object and modify the appropriate attributes or rules associated with the object.

One of the traditional limits of early expert systems was the inability of the user to operate in a "mixed initiative" mode. Once backward chaining was started, it was difficult to interrupt the decision-making process to volunteer information or to change the course of an interaction. One source of this inflexibility is that early expert systems use a small number of representation and control strategies, such as rules and backward chaining. As suggested in earlier section, a hybrid AI development system can support both traditional frame based representation and rule-based reasoning with interactive browsing and flexible modification of knowledge base content.

Natural Interaction through Graphics and Active Objects. In addition to using frames to describe objects themselves, Figure 3 suggests that graphical descriptions may



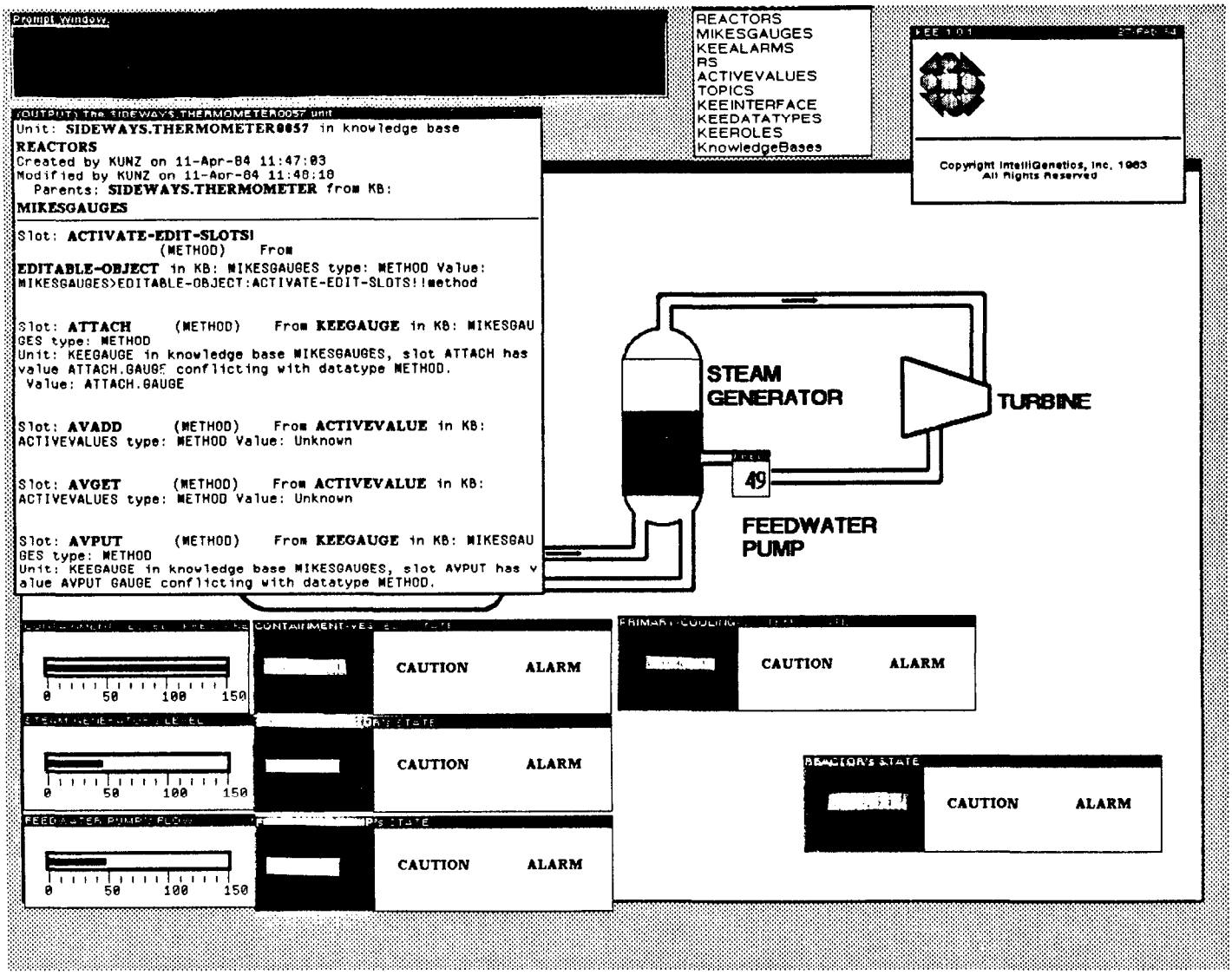
Overview of the design of an alarms processing system within the KEE knowledge-based system development environment.

Figure 6.

help to explain the contents of propositional representations. Figure 5 shows another use of graphics to represent a control panel for REACTORS. This panel shows an iconic representation of some of the objects of the domain, illustrating their spatial relations.

Engineers characterize alarm conditions from several perspectives, including alarms associated with individual component reactor parts, with abstractly described reactor subsystems, and with the plant itself. For uniformity of presentation to the KB user, the REACTOR'S control panel includes a set of normal, caution and alarm state indicators that describe the operating status of plant component parts; another set for describing status of reactor subsystems; and a single group of indicators for describing plant status.

Deepening the Representation. In comparison with early AI systems, problem-solving can be made more flexible and robust by building systems that represent and reason using knowledge of the structure and function of the domain. One of the sources of the power of expert systems is that they can perform problem-solving in a context-dependent way because rules explicitly describe the context in which they are applicable. However, they depend upon the designer to articulate each important problem-solving context. In addition, the contexts are described as associations by their designers, and those associations normally lack any underlying principle to guide their creation by designers, use by the sys-



Description of an example object in the REACTOR'S KB.

Figure 7.

tem, or understanding by users. Experiential knowledge has compiled away the facts of structure and the principles of function, replacing them with a large number of particular applications in different contexts. With an appropriately broad and powerful set of representation tools and knowledge bases, designers will be able to represent the structure and the function of domains. Representations will be more concise when a small number of important principles are used for problem-solving in many contexts. Description of the knowledge of the structure and function of domains supports systematic organization and acquisition of the knowledge of a domain, since knowledge acquisition is guided by the principles of the domain rather than being *ad hoc*. The user can use such a system as an "intelligent encyclopedia" for browsing and experimenting with the components of a modeled system (*e.g.*, by being able to identify what happens if a low-impedance voltmeter is connected to a thermistor, or if two particular DNA structures are joined). If the system does not know about a particular object, such as a low-impedance voltmeter, the user can create a description of one by appropriately specializing the description of a generic voltmeter.

Example Scenario

Now consider the way that a user can interact with REACTORS. This example scenario shows the flexibility of control of the operation of the knowledge-based system. In this example, the application system designer has made the attributes of the reactor parts and subsystems "active," so that a set of LISP methods is invoked when those attribute values are changed. Numbered items refer to the sequence of events shown in Figure 6.

1. To analyze a particular problem, such as loss of coolant water, the user identifies the water level of the steam generator in the control panel (See the object labeled "STEAM-GENERATOR" in Figure 5) and uses the mouse to reduce its water level. Active graphics provide visibility as well as control over the knowledge base.
2. When the level of the STEAM-GENERATOR device is changed, active value methods are invoked automatically. Active values, or demons, provide data-driven control of the system operation.
3. Active value methods have been written to update the graphics icon to show the new parameter value and to store that new value in the underlying symbolic representation. In addition, when an alarmed parameter value is changed, the alarm active value method checks whether an event has occurred, or whether the parameter has changed state (*e.g.*, from its normal to its low level). Significant events are recorded in the local rule system working memory, and the rule system is invoked to find the implications of events. Active values, graphics, frame-based representation, rule-based reasoning, and LISP methods are hybridized to support each other.
4. Some rules in the application KB simulate the effects of significant events. These simulation rules are invoked (in a forward direction), if appropriate, to propagate the effects of a significant event. When an event occurs, such as the coolant level becoming low, a rule can check preconditions and, if they are met, store the name of the new state in an attribute of the appropriate object. Reporting a state-change event in turn invokes the active value mechanism, which reports the new state value to the user in the graphics display. In addition, analysis rules are invoked (in a backward direction) to analyze the new plant state. Ready interaction between forward and backward chaining allows a mixed bottom-up and top-down analysis of data.
5. While rules are being invoked, the user becomes curious about the state of the STEAM-GENERATOR Browsing, she selects this object with the mouse in the control panel and brings up a command menu. She uses the mouse to request a display of the symbolic structure of the object and reviews it. (See Figure 7.) While browsing, she can change the value of any of the attributes of this object or ask the object to perform any of the actions of which it is capable, such as displaying itself, identifying itself, or analyzing its behavior. When finished browsing, she can continue the rule interaction. User understanding of the domain and the domain model's facilitated by flexible control, which allows user examination and change of the KB at any time while running the system.

The alarm system design allows physical parameters of component parts to describe the values that designate their normal operating ranges. As shown in Figure 7, for example, the STEAM-GENERATOR frame has an attribute called level, which in turn has an alarm limits property that specifies the low, normal, and high ranges for the quantitative values of this physical parameter.

Hybrid AI Development Systems

LOOPS (Stefik, 1983) introduced a model of a hybrid AI system. KEE is a hybrid AI development system that integrates frames for representation of static knowledge, rule-based reasoning, graphics, and active values (Kehler, 1984). Hybrid systems have been developed in response both to the availability of powerful LISP-based workstations and, as discussed in the Introduction, to the goal of providing rich environments for development of knowledge-based systems.

Frame-Based Representation and Rule-Based Reasoning

Frame-based representation allows users to describe objects in their domains. Frames can describe attributes that are either declarative or procedural in form, and attributes can be inherited from one object to its descendants according to semantically precise rules of inheritance. Inheritance provides one way to specify semantic proximity between objects. Another way to specify semantic proximity between the attributes of objects is to specify the relations among rule preconditions that refer to attributes of objects.

MYCIN and the large number of systems that it inspired have demonstrated the power of rule-based reasoning for analysis of problems in complicated domains. Rules are easy to use, and they both allow and force a focus on decision making and simulation. One of their greatest values is that they are an effective *interlingua* that can be used by the designer, the user, and the computer. Thus, production rules suggest easy and appropriate explanation facilities, such as the rule graph shown in Figure 3.

Frame-based representations allow users to describe the abstract and concrete objects of their domains, and rules allow users to describe heuristics and procedural knowledge of a domain. Anderson discusses production rule and frame-based representations extensively, and he argues that they are complementary from a cognitive science point of view because production systems are models of skills, and frames are patterns for describing and recognizing recurring sets of features (Anderson, 1980).

LISP

LISP is the underlying system that has been used almost universally by AI systems developers, although PROLOG has become popular, and some systems are and will be developed using other languages. Some AI development systems encourage users to access the underlying system language whether it is LISP, PASCAL, ADA, or something else and provide facilities so that access is direct and relatively easy. Other systems attempt to insulate users from the underlying system by providing large numbers of specialized functions to support users of rules or frames. In this case, users must develop or buy the consulting services of trained programmers when the built-in features fail to meet some need.

Unskilled users will often be intimidated by any programming language, and procedural languages are inherently opaque in their operation. However, only a full programming language provides access to the full power of a computer. Thus, the issue is how to trade off that complexity and power. The current methodologies of representation, rule-based reasoning, and graphics-based explanation are sufficiently powerful that many users will not need constant access to the underlying programming language. In a hybrid development system, the system kernel that supports representation and reasoning can be relatively small and general when users are given the responsibility for developing specialized computational functions in an underlying programming language. In addition, the users' process of developing appropriate specialized features can be aided by identifying interesting generic features from some library and modifying them for particular purposes. Finally, the process of analyzing different representation and manipulation techniques contributes to the user's understanding of the problem. In contrast, it is difficult to design a simple and efficient system that will provide almost all of the features that most users will ever need, and users are constrained both intellectually and technically when they have limited ability to explore alternative problem representation and manipulation approaches.

Graphics

Graphics in LISP-based workstations are implemented as mouse-sensitive icons in windows. Graphics can facilitate the explanation of problem structure and behavior directly, as shown in Figures 3, 4, and 7. Such graphical descriptions can convey far more information than is readily presented in a simple teletype-style computer terminal. Connotative information can be conveyed in graphical representations, such as the information conveyed to human users in the shape of objects shown in Figure 5. Schematic descriptions of physical objects provide users with information about the model that underlies the schematic. In addition, the elements of the schematic can be active. For example, as shown in Figure 5, the user can actually view the fluid level in the steam generator because it moves up and down as the corresponding attribute value is changed. In addition, as established, the user can change the fluid level using the mouse, and the underlying control mechanism can then both propagate the effects of change through the system and display the results of those changes graphically.

Active Values

Active values are attributes of objects that have attached demons that are invoked when their values are accessed or stored. Active values integrate the behavior of representation and graphics by defining methods that update the iconic description of an attribute value whenever its value is changed. Active values allow simple operations to be performed automatically, in a data-directed way. For example, the alarm filtering process discussed earlier was implemented using active values. They can be used to integrate representation, reasoning and graphics so that the appropriate graphical and symbolic representations are updated whenever an active attribute value is changed, whether by a successful rule invocation, by effect of a method, by the user changing a value of an attribute manually, or by the user changing an attribute value in an icon using the mouse.

Object-Oriented Programming

Object-oriented programming is the design principle, pioneered in SMALLTALK, that descriptive and procedural attributes of an object should be associated directly with that object. Object-oriented programming can thus be highly modular. Since each object has its own procedural characteristics, it can perform local actions such as display or modify itself, and it can both receive information from and return information to other objects. Object-oriented programming is the principle that is used to unify the major KEE methodologies. Design and use of each of these methodologies—representation, rule-based reasoning, LISP, graphics, and active values—is object-oriented.

Object-oriented programming provides a simple principle for unifying the major methodologies. Frames can be fully object-oriented if they contain both descriptive and procedural attributes. The MYCIN-style of rule premise, an ⟨OBJECT - ATTRIBUTE - VALUE⟩ reference, is explicitly object-oriented, and rules with this object-oriented syntax can refer explicitly to the attributes of frames. LISP methods are object-oriented if they are associated with objects and avoid side effects. Graphical icons are inherently object-oriented because they represent objects.

When rules and frames are integrated, attribute values assigned by inheritance can be tested within rules. Rule premises inherently define semantic proximity between attributes of different objects. Thus, when integrated with frames, rules give a very general way for defining arbitrary semantic relations in a frame-based representation system.

Frame-based attribute inheritance normally supports relations between objects, such as Instance-of and Member-of, while pure rule-based systems must specify these common relations with rules or rule premises. Thus, in comparison with pure rule systems, it is possible to reduce both the number of rules and the average number of premises per rule by attaching rules to objects within an object-oriented inheritance hierarchy. Finally, in a pure rule-based system, the interpreter must test screening premises in large numbers of rules to find the smaller number of rules that may be applicable. In contrast, when rules are associated with the objects to which they refer, rule-based reasoning systems need consider only those rules that are known to be potentially applicable to the specific object.

Applications Development Process

Our understanding of an integrated methodology for using multiple programming methodologies in a hybrid tool environment has been evolving over the past year. This section discusses our present understanding of the problem and identifies some techniques we use to control the development process.

The primary problem in developing an AI application system is to identify the work to be done, or the total problem that is to be solved from the point of view of the organization that has the problem. Typically, several contributors will participate in the problem solution, possibly including several people with different skills, procedures for organizing people, equipment, instrumentation, traditional computer systems and knowledge-based systems. The work to be done can often be characterized as the making of a series of decisions.

When viewed from a broad perspective, one nearly universal characteristic of interesting problems is that in the beginning, they are poorly specified. If they were clearly specified, most of them probably could and would have already been solved. Thus, two overriding objectives of the knowledge-based system development process are to develop a problem definition that is precise enough that it can guide system development and to select tools that are flexible enough that the system can be changed in response to an evolving understanding of the problem. This position is analogous to one taken by (Sheil, 1983). Ideally, the tools are communicative, so that operation of the prototype systems will help the user to understand the problem better.

One overriding theme of our approach to developing knowledge-based system applicationsis to focus separately on the work to be done and on the tools to do the work. Initially, a tension exist between these two foci. Solution-oriented people will quickly seize on some approach, such as simultaneous equations in FORTRAN, or MYCIN-style rules, or some heuristic search algorithm in LISP. Problem-oriented people quickly ask what the problem is, and then point out that the problem is not yet well understood. The most successful projects exploit this tension to find problems that have both value and solutions. Our principal method of finding the match between problems and solutions is to develop a series of prototypes that can be used incrementally to develop both the problem specification and the solution concurrently.

Flexibility is the hallmark of AI-based system development tools. Flexibility supports rapid prototyping, reformulation of system designs, experimentation with the application of realistic data to the prototype systems, potential integration of multiple problem-solving techniques, and consciously opportunistic development strategies. It is not that any particular system could not be built in any given programming language or using more traditional programming techniques. Rather, traditional systems force the user to fit the problem to the tool. Hybrid systems support a number of different approaches and integration of the different programming approaches to encourage discovery of both alternative system specifications and solutions.

Thus, goals of the knowledge-based system development process include:

- Rapid prototyping, to explore both the problem and the solutions:
  - Find a "quick win" that carries an important piece of the intended functionality of the system. This quick win should use conservative techniques and should make the power of the technology as visible as possible.
  - Beyond the quick win phase, there should be a major technological vector that can incrementally add to the value of the system. This second phase can involve a high risk technology.
  - Use mock-ups of peripheral features so that the real purpose of the demonstration can be achieved quickly.
- Declarative representations, to make the problem and its solutions visible both to the designer during exploratory programming and later to the user
- Early focus on the user of the knowledge-based system:
  - Identify and support ways that the system will be integrated into the using organization.
  - Identify and support explanation that will help system users.
- Specialization of existing representations and methods:
  - Identify ideas and computer code to modify and exploit.
  - Design generic objects that can be specialized and refined.
  - Specialize and refine generic objects
  - Don't redesign unless necessary
- Exploratory programming:
  - Try multiple solutions to explore alternative ways to do the work.
  - Pick and choose different methods for different purposes.

We recognize that this approach is at fundamental odds with traditional software engineering methodology, and this difference gives us pause rather than pride. Software engineering techniques are well enough understood that they have been scaled up to organizational mechanisms to produce software products. There is not yet similar experience with AI systems. A few AI-based applications "products" are now in limited use. Issues of maintenance, documentation, customer support, and validation are just starting to be addressed a serious manner in different applications areas. An open question is when it is better to deliver AI systems in the form in which they were developed and when to use the initial systems as detailed and 'active' specifications of systems to be converted into deliverable products using traditional software engineering methodology.

Conclusions

Starting from the premise that AI applications problems are initially both ill-structured and complicated, this article discusses techniques that can be used to help the designer to specify problem solutions by developing a series of prototypes that show increasing precision in their specification and that explain both the problem description and the solution method to developers and users. Ultimately, the problem becomes better defined, but it must continue to be sensitive to the context of each individual case, and it often remains complex. Traditional AI development systems emphasize a single representation and programming methodology, such as frames, rules, or logic programming, as a methodology that is natural, efficient and uniform. The applications we have developed suggest that naturalness, efficiency, and flexibility are increased by trading uniformity for the power that is provided by a small set of appropriate programming and representation tools.

Our application development methodology uses a strategy based on separate and explicit focus on the work to be done and the methods for doing the work. The goal of the methodology is to achieve both an effective specification of and a solution to a problem. The development methodology emphasizes working toward problem specification and solution simultaneously by specializing generic objects and behaviors and by exploiting a hybridized collection of AI development tools.

Viewed abstractly, this methodology emphasizes developing a series of "expert support systems." The development of such systems may ultimately lead to traditional expert systems, or the support systems may continue as tools for exploration of complicated problems by experts. The development prototypes, or expert support systems, are computational models of the structure, function, and behavior of modeled systems. Their use helps the expert to articulate and to understand the domain and the criteria for decision-making within the domain. By using a variety of representation and reasoning techniques, the developer can explicitly attempt to identify sources of power for performing the work to be done, and often that power lies in complementary use of appropriate representations of the domain, reasoning knowledge and strategies, user interfaces, and specialized computational models.

Expert support systems seem to derive power from three fundamental sources. First, they need and exploit several related AI methods. These methods must be integrated so that they support each other. Useful methods include frame-based knowledge representation with inheritance, rule-based reasoning, LISP, graphics, and active values. Second, expert support systems depend upon powerful explanation to make the system structure and behavior available to the developer and later users. Useful explanation features include complementary symbolic and schematic description of the structure of the domain and the behavior of systems in the domain and description of the way that rules are used during the process of decision-making. Finally, expert support systems derive power from allowing the developing expert to use the system in many ways, including creation of new concepts, rules, and graphics; description of the contents of the system; running of the computational model; interactive running of the model; and browsing about and changing its content.

References

Anderson, J.R., P.J. Kline, and C.M. Beasley. (1979) A general learning theory and its application to schema abstraction. In G.H. Bower (Ed.), *The psychology of Learning and Motivation.* New York: Academic Press.

Anderson, J.R. (1980) *Cognitive Psychology and Its Implications.* New York: W.H. Freeman.

Chandrasekaran, B. (1983). Towards a Taxonomy of Problem Solving Types. *AI Magazine*, Vol. 4, No. 1, 34-37.

Kehler, T.P. and G.D. Clemeson (1984) An application development system for expert-systems. *Systems and Software 34* :212-224.

Shiel, Beau (1983) Power Tools for Programmers. *Datamation.*

Stefik, M., D.G. Bobrow, S. Mittal, and L. Conway (1983) Knowledge Programming in LOOPS. *AI Magazine*, Vol 4., No. 3, 3:14.