KNOWLEDGE MANAGEMENT

The introduction by Allen Newell in 1982 of the knowledge-level principle (1) has cast a whole new light on the definition of the knowledge management discipline. According to this principle, knowledge level represents the highest level in the description of any structured system. Situated above the symbol level and independent from this, it describes the observed behavior of the system as a function of the knowledge employed, independently of the way this knowledge has been represented at the symbolic level. As Newell says: "The knowledge level permits predicting and understanding behavior without having an operational model of the processing that is actually being done by the agent" (1, p. 108). An arbitrary system is interpreted as a rational agent that interacts with its environment to attain, based on the knowledge it has, a given goal in the best way. From the viewpoint of a strict knowledge level, it is considered a sort of "black box" to be modeled on the basis of its input/output behavior without making any hypothesis about its internal structure. To sum up, the knowledge-level principle emphasizes the "why" (i.e., the goals), and the "what" (i.e., the different tasks to be accomplished and the domain knowledge) more than the "how" (i.e., the way of implementing these tasks and of putting this domain of knowledge to use).

The emergence of this principle has transferred interest in the knowledge management field from the pure representational aspects to the modeling aspects, that is, a shift from the production of tools for directly representing the knowledge a system uses to that of tools for building up models of the system's behavior in terms of that knowledge. A well-known example of this tendency is a European result, the Knowledge Acquisition and Design Structuring (KADS) methodology (2,3), with its developments and derivatives.

A fundamental step in the KADS approach is the setup of a general conceptual model of the system that an observer (a knowledge engineer) creates by abstracting from the problem-solving behavior of some experts. According to the knowledge principle, the conceptual model does not include any detailed constraints about the implementation level. This last function is specific for the design model, which can be considered a high-level system description of the final knowledge-based system (KBS), and which represents the transformations to be executed on the conceptual model when we take into account the external requirements (e.g., specialized interfaces, explanation modules, etc.). The conceptual model is built up according to a four-layer structured approach. Each successive layer interprets the description given at the previous layer. The first layer (category of knowledge) is concerned with the static domain of knowledge, the domain concepts and their attributes, the domain facts, the structures representing complex relationships etc. Static knowledge can be viewed as a declarative theory of the domain. A second type of knowledge (inference layer) is concerned with the knowledge sources and the metaclasses. A knowledge source is defined as an elementary step in the reasoning process (an inference) that derives new information from the existing source. KADS presupposes the existence of a set of canonical inferences such as abstraction, association, refinement, transformation, selection, and computation. Metaclasses describe the role that a group of concepts plays in the reasoning process (e.g., observable, hypothesis, solution). The third layer contains knowledge describing how inferences are combined to fulfill a certain goal, that is, how to achieve operations on metaclasses. The most important type of knowledge in this category is the "task." A task is a description of a problem-solving goal or subgoal, for example, "diagnose a patient with these particular symptoms." The fourth category of knowledge is the strategic knowledge, that settles the general goals relevant for solving a particular problem. How each goal can be achieved is determined by the task knowledge. The software counterpart of this structured methodology is a set of tools (a workbench) including, for example, a domain text editor to analyze interview transcripts, a concept editor for the domain layer modeling, an inference structure editor, a task model tool supporting the identification of the structure of a particular problem solving task by decomposing the task and establishing the relevant task and domain features, libraries, graphical tools, etc. At the top, an advice and guidance module controls the general development of the KBS and provides advice on the basis of the KADS methodology. KADS tools are commercialized, for example, by the French ILOG company, also established in the United States. Recent developments are concerned with, inter alia, establishing an advanced formal modeling language (ML^2) to describe the conceptual model and with some standardization work (Common KADS).

COMMET (4) is a methodology that has some points in common with KADS. It is based on the principle that the knowledge-level description of expertise includes three major components: the model perspective, the task perspective, and the method perspective. In a more specific context of knowledge acquisition, we can mention PROTÉGÉ-II (5). This is a knowledge-acquisition shell that uses problem-solving methods to drive the modeling of some specific tasks. For example, given a set of symptoms for a faulty device, like manual observations and instruments readings, produce a diagnosis and a remedy. Method configuration in PROTÉGÉ-II is carried out by using a library of basic building blocks (black boxes) called "mechanisms."

One of the main attractions of this new, structured and analytical approach to knowledge management is that all of the methodologies based implicitly or explicitly on the knowledge-level principle embrace the idea that the setup of KBSs is facilitated by developing libraries of reusable components. These pertain mainly to two different classes: reusable ontologies, that is to say, (normally tangled) taxonomies defining the concepts (important notions) proper to a given domain and their relationships (6), and reusable problem-solving methods, which define classes of operations for problem solving. In this last context, we can mention Chandrasekaran's work (7). Chandrasekaran was one of the first scholars to suggest developing reusable components under the form of "generic tasks." A generic task defines both a class of application tasks with common features and a method for performing these tasks. In this respect, these new knowledge management methodologies have many points in common with the work accomplished within the ARPA Knowledge Sharing Effort (8). A concrete product of this work is KIF, a general, declarative specification language for Knowledge Interchange Format, that has declarative semantics and provides, among

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

other things, for asserting arbitrary sentences in the first-order predicate calculus, expressing metaknowledge, and representing nonmonotonic reasoning rules (9).

An additional manifestation of this general tendency toward generalization, abstraction, and reuse is the activities aimed at constructing general and reusable "corporate memories." In the recent years, knowledge has been recognized as one of the most important assets of an enterprise and a possible success factor for any industrial organization if it is controlled, shared, and reused effectively. Accordingly, the core of the organization can be conceived of as a general and shared corporate memory, that is, an on-line, computer-based storehouse of expertise, experience, and documentation about all the strategic aspects of the organization (10). Then the construction and practical use of corporate memories becomes the main activity in the knowledge management of a company, a focal point where several computer science and artificial intelligence disciplines converge: knowledge acquisition (and learning), data warehouses, database management, information retrieval, data mining, case-based reasoning, decision support systems, and querying (and natural language querying) techniques.

The knowledge-level revolution has been of fundamental import for the methodological renovation of the knowledge management discipline. However, from a more practical point of view, the concrete results have not been so immediate as were expected and, after a peak of interest at the beginning of the nineties, all of the issues concerning, for example, knowledge sharing and reuse now have attained a more relaxed cruising speed. There are in fact several factors that can contribute to delaying the fulfillment of all of the benefits we can expect from applying the new methodologies. For example, from a theoretical point of view, some methodologies that refer to the knowledge-level principle in reality run counter to Newell's approach because the structure they impose on the knowledge is a function of "how" a specific class of applications is implemented and dealt with and the models they produce are then valid only in a very specific context. On a more pragmatic level, reuse can be very difficult to obtain because there is often a significant semantic gap between some abstract, general method and a particular application task. Moreover, discovering and formalizing a set of elementary tasks in a way that is really independent of any specific application domain is a particularly difficult endeavor which encounters all sort of embarrassing problems, ranging from the difficulties in defining the building blocks in a sufficiently general way to the ambiguities about which aspects (the model or the code) of the blocks can really be reused. This explains why a (not trivial) number of knowledge-level proposals are still theoretical and are characterized by a limited or no implementation effort.

But the main problem of these new methodologies based on a pervasive modeling approach is linked with the fact they forget that the core technology for knowledge management is still represented by knowledge representational (and processing) techniques. To be concretely used, the building blocks, the generic tasks, the reusable modules, and the shareable ontologies must eventually be formalized by using one or more of the ordinary knowledge representational techniques, rules, logic, frames, or whatever. Forgetting this common sense rule to emphasize the modeling and methodological virtues of the knowledge principle can lead, for example, to rediscovering (downgraded) versions of traditional semantic networks under the form of "concept maps" or to producing a further, paper-implemented catalogue of generic axioms. In this article, knowledge management is described essentially as an application of the usual knowledge representational (and processing) techniques. Creating and using large corporate memories requires, first of all, that the knowledge can be represented, stored, and computer-managed realistically and efficiently.

THE TWO MAIN CLASSES OF KNOWLEDGE REPRESENTATION (AND PROCESSING) SYSTEMS

"Knowledge is power," according to the well-known slogan spread by Edward Fegenbaum. More precisely, Fegenbaum stated that: ". . . the power . . . does not reside in the inference method; almost any inference method will do. The power resides in the knowledge" (11, p. 101). Even those researchers (e.g., the advocates of a strictly formal logical approach), who do not appreciate this way of reducing the importance of the algorithmic aspects of the artificial intelligence (AI) endeavor, will agree on the fact that knowledge representation is probably the key problem in AI. One could object that some knowledge about a particular problem domain is, in fact, embedded in every computer program. The simplest word processor contains a considerable amount of knowledge about formats, characters, styles, editing techniques, and printing. However, in ordinary computer programs, knowledge is not represented explicitly and cannot be smoothly reconstructed, extracted, or manipulated. This contrasts strongly with the AI approach, at least in its symbolic form (see later), where the importance (from a strictly quantitative point of view) and the complexity of the notions inserted into a machine that lead it to behave in some sort of "intelligent" way implies that these notions (the knowledge) must be studied, represented, and manipulated in themselves. Then the aim of AI is to produce descriptions of the world so that, fed into a machine, it behaves intelligently simply by formally manipulating (knowledge management) these descriptions (12). If we renounce any strong hypothesis about the final achievements of AI, that is, if we admit that AI will at best simulate some external results of human intellectual activities, but not the inner mechanism itself, the emphasis on knowledge representation becomes one of the most important criteria to justify identifying AI with a well-defined and separate subfield of computer science.

Now the problem is how to represent formally the knowledge that must be supplied to the machine, knowledge that we can think of as formulated initially by some sort of verbal representation. A useful, if somewhat simplified, classification consists of isolating the following two main groups of knowledge representational techniques (all sort of mixed approaches are obviously possible):

- Techniques that follow the classical, symbolic approach. They are characterized by (a) a well-defined, one-to-one correspondence between *all* of the entities of the domain to be modeled and their relationships, and the symbols used in the knowledge representational language; and (b) by the fact that the knowledge manipulation algorithms (inferences) take this correspondence into account explicitly.

- Techniques that we can define as biologically inspired, like genetic algorithms or neural nets. In these techniques, only the input and output values have an explicit, one-to-one correspondence with the entities of a given problem to be modeled. For the other elements and factors of the problem, (a) it is often impossible to establish a *local,* one-to-one correspondence between the symbols of the knowledge representational system and such elements and factors; (b) the resolution processes are not grounded on any explicit notion of correspondence; (c) statistical and probabilistic methods play an important part in these resolution processes.

Biologically inspired techniques are dealt with in depth in separate articles of the encyclopedia. See, for example, the MACHINE LEARNING article. In the next section, then we limit ourselves to evoking briefly the main properties of neural networks and genetic algorithms, also briefly mentioning the fuzzy logic approach that is often associated with the two previous techniques. Expressions like “soft logic” or “soft programming” are often employed to designate the union of these three unconventional techniques. The remaining sections of the article are devoted totally to the symbolic approach.

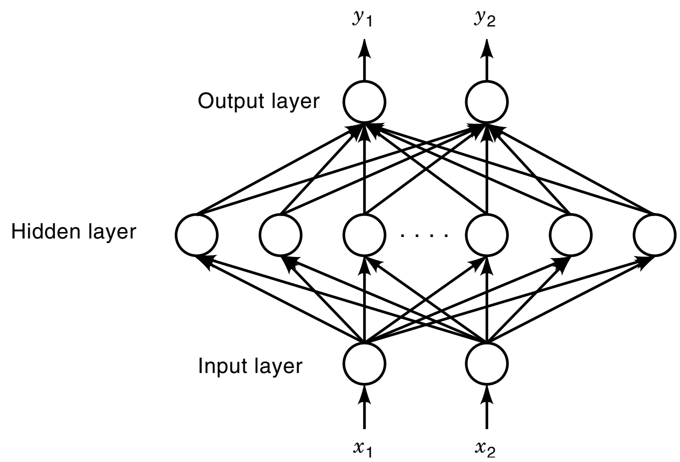

Figure 1. A typical three-layer neural network, including an input, hidden and output layer. Each neuron in each layer is connected with all the neurons of the previous layer; the strength of the connection (“synapsis”) between two neurons is given by an associated weight w.

THE BIOLOGICALLY INSPIRED APPROACH

Neural Networks

After a period of oblivion due to the demonstration by Minsky and Papert (13) of the shortcomings inherent in the pattern-recognition capabilities of a particular class (perceptrons) of first-generation neural networks, neural nets again became a very fashionable subject study at the beginning of the 1980s. More than loosely analogous with the organization of the brain—in this last contest, only the (very simplified) concepts of “neuron” and “synapsis” have been preserved—the biological foundations of neural networks reside in the self-organizing principles characteristic of living systems. When a threshold number of interconnections (synapes) have been established between a set of neurons and if the network has been carefully programmed, a form of self-organizing activity appears that allows an external observer to affirm that the network learns. For example, it learns to associate a pattern with another, to synthesize a common pattern from the a set of examples, to differentiate among input patterns, where pattern is understood as its more general meaning. See Refs. 14 and 15 for a detailed account of neural networks theory.

A neural network is generally composed of several layers, in which any number of neurons can be present in each of the layers. Figure 1 shows a typical three-layer network: The first layer is the input layer, the last the output layer, and the layer in between is the hidden layer. Each neuron in each layer is connected with all the neurons of the previous layer. All of the neurons act as processing units. Each neuron maps the multidimensional inputs received from all of the other neurons (processing units) situated in a lower layer (or some external stimuli) to a one-dimensional output. Then the activation level of a generic neuron i in layer n is determined in two steps (see Fig. 2). First, we calculate the weighted sum of the j inputs to this neuron:

$$s_{n,i} = \sum_j w_{n,i,j} a_{n-1,j}$$

where $a_{n,i}$ is the output (the activation level) of the neuron i in layer n, and $w_{n,i,j}$ is the weight associated with the connection between the neuron i in layer n and neuron j in layer $n - 1$, that is, the strength of this connection. The weights can be either positive, tending to excite the receiving neuron, or negative, tending to inhibit the receiving neuron. An important point is that the activation level of each neuron must be bounded, and then permitted to vary between values that can be, for example, 0 and 1.0. This is linked, inter alia, with the fact that the activation level of an artificial neuron (called sometimes a neurode) is intended to simulate the frequency of neuronal firing in an animal. Given that negative frequencies have no meaning, no negative values are usually admitted for the activation levels. Moreover, the values are

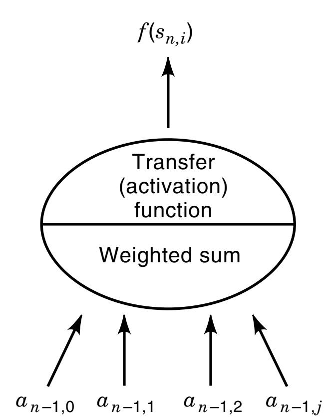

Figure 2. The activation level of a generic neuron is determined in two steps. First, we calculate the weighted sum of the inputs to this neuron. Secondly, a transfer or activation function is applied.

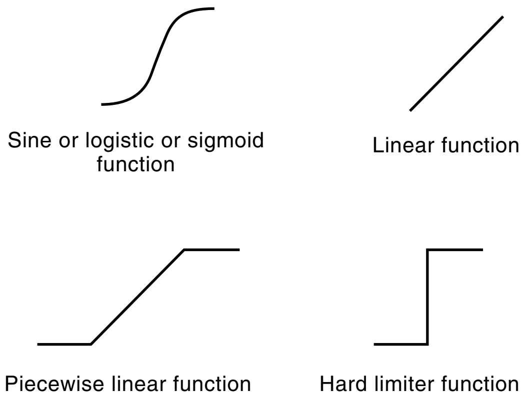

Figure 3. Four possible transfer or activation functions considered in an $(-1, 1)$ domain.

bounded because biological neurons have a maximum firing frequency beyond which they cannot go. Then the final activation level of neuron i in layer n is given by

$$a_{n,i} = f(s_{n,i})$$

where f is the transfer function or activation function. See Fig. 2 again.

The most commonly used transfer function is the sine, or logistic, or sigmoid (because of its S shape) function, but many other functions are possible. The four transfer functions normally mentioned in a neural network context are represented in Fig. 3. The equation of the linear function is, obviously, $y = x$. If a linear transfer function is used, then $a_{n,i} = s_{n,i}$. The equation of the sigmoid is $y = 1/(1 + e^{-x})$ in the interval (0, 1), and it is $y = \tanh(x)$ in the global interval $(-1, 1)$. The piecewise linear function has a linear behavior in a given interval of x, and it is squared outside this interval. For y limited to a (0, 1) interval, we could have, for example, $y = (1/6) * x + 0.5$ for $(-3 < x < 3)$; $y = 1$ for $(x \geq 3)$; and $y = 0$ for $(x \leq -3)$. For y spanning the whole $(-1, 1)$ interval, we could have $y = (2.0/4.0) * x$ for $(-2 < x < 2)$; $y = 1$ for $(x \geq 2)$; and $y = -1$ for $(x \leq -2)$. The hard limiter function has only a historical significance, associated with the old perceptron era. In the interval (0, 1), it takes a value $y = 1$ when $x \geq 0$. Otherwise $y = 0$. In the interval $(-1, 1)$, it takes a value $y = 1$ when $x \geq 0$. Otherwise $y = -1$.

Many alternatives have been proposed with respect to learning techniques. We mention briefly the backpropagation method, probably the most widely used learning technique. It is based on the principle of adjusting the weights using the difference, for a given distribution (pattern) of input values to the network, between the desired activation levels for the neurons of the output layer and the levels really obtained. Then using a training set composed of couples of input-output patterns, the weights are cyclically modified so that the differences are eventually minimized according to a least-squares approach. In the multilayer case considered in this section and simplifying greatly the real situation for comprehensibility—we have to solve equations that have this general form:

$$\min_{w} 1/m \sum_{k=1}^{m} [y_k - f(x_m, w)]^2$$

For a given input pattern of the training set, indicated here simply by x_{pi}, we have to minimize the average squared error between the corresponding output pattern (the desired values y_k) associated with the m neurons in the output layer and their actual activation values. As already stated, these values result from the repeated application of an activation function f to some values s which depend generally on both the input values to the network, x_{pi} in this case, and the weights w: w is the parameter to be adjusted (the variable). Finding the minimum of the above expression implies finding the first derivative of f. This is really simple to calculate if f is a sygmoid, expressed as $f(x) = 1/(1 + e^{-x})$ (see above), its derivative is simply $f(x)\,[1 - f(x)]$. The backpropagation activity begins with calculating the errors for the output layer. Then the cumulative error is backpropagated from the output layer to the connections between the internal layers to the input layer and is used to reassign the weights. The correction of the weights $w_{n,i,j}$ for the connection between neurons in layer n and neurons in layer $(n - 1)$ uses an error gradient, which is a function of the first derivative of f evaluated at layer n, of the total signal error backpropagated from the subsequent layer $(n + 1)$ and of the weights of the connections between layer $(n + 1)$ and layer n.

Some advantages of the neural network approach and the conceptual differences with the symbolic approach are well illustrated by the following example derived from (16). It represents the neural network solution to a well-known problem in robotics, the inverse kinematic problem. It can be schematized as in Fig. 4, where a robotic arm made of two linear segments of fixed length l_1 and l_2 can modify the joint angles θ_1 and θ_2 and move in a two-dimensional plane. The problem consists of finding the values of θ_1 and θ_2 for some expected positions (x, y) of the free end point of l_2. From Fig. 4, it is easy to see that the Cartesian position of this end point is given by

$$x = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) \qquad (1)$$
$$y = l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2)$$

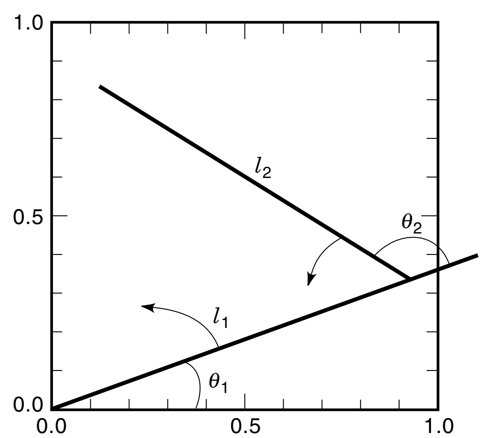

Figure 4. The inverse kinematic problem. A robotic arm made of two linear segments of fixed length l_1 and l_2 that can modify the joint angles θ_1 and θ_2 and may move in a two-dimensional plane. The problem consists of finding the values of θ_1 and θ_2 for some expected positions (x, y) of the free endpoint of l_2.

Using equivalencies like

$$\cos(a-b) = \cos a \cos b + \sin a \sin b$$
$$\tan x = \sin x/\cos x$$

we can express Eq. (1) in terms of θ_1 and θ_2 to obtain

$$\begin{aligned}\cos\theta_2 &= \frac{(x^2+y^2-l_1^2-l_2^2)}{2\,l_1 l_2}\\ \theta_1 &= \arctan\left(\frac{y}{x}\right) - \arctan\left[\frac{l_2\sin\theta_2}{(l_1+l_2\cos\theta_2)}\right]\end{aligned} \tag{2}$$

A concrete use of equations like Eq. (2) requires, in practice, a cumbersome manipulation of predefined tables of coordinate transformations. Moreover, the use of the tables may be ineffective for minimal changes in the robotic structure resulting from natural or accidental causes. Then the neural network approach described in Ref. 16 uses a three-layer network like that of Fig. 1, where the two neurons of the input layer represent the (x, y) Cartesian coordinates of the free end point, the single hidden layer is made up of 32 neurons, and the two neurons of the output layer represent the θ_1 and θ_2 values. A backpropagation learning algorithm is used. The results of Ref. 16 show that a precision of about 100% (with a predefined tolerance of 0.025) is already obtained after less than ten training examples. This means that, when the network is presented with additional and unseen examples of end-point coordinates, it can compute the corresponding joint angles with a precision that is well in agreement with the predefined error. The advantages with respect to conventional computational schemes are evident. Neural networks can learn to transform from Cartesian coordinates to angles from examples only, without any need to derive or program the solution of inverse equations. Natural or accidental changes in the topology of the device are automatically taken into consideration by the network. Using a neural network approach, the solution space does not need to be very precisely defined, that is, the robot learns to behave in a more approximate environment. Of course, the astonishing success of this particular type of application—which can be reduced to a pattern recognition problem, a domain where the utilization of neural networks is particularly recommended—must not be overestimated. The symbolic approach can cover, in fact, a number of possible domains of utilization that are surely more important generally than the number of domains where some biologically inspired approach is particularly appropriate.

We conclude this section with some general remarks. The example considered in the previous paragraphs is a good illustration of the differences between the symbolic and biologically inspired approach. In a symbolic approach, the situation of Fig. 4 is represented by Eqs. (1) and (2). The inference procedure consists of solving Eq. (2) for a given couple (x, y), for example, by using predefined tables. The same situation is represented in a neural network approach by a network like that of Fig. 1 where a correct distribution of weights has already been learned. The inference procedure corresponds again to running the network for a given couple (x, y). In the symbolic approach, there is a very precise, one-to-one correspondence between entities to be modeled and symbols. For example, the first segment of the arm corresponds to the symbol l_1, the first angle to θ_1, etc. In the second case, the knowledge representation is distributed and linked with the interaction, at a given instant, between the topology of the network and a given distribution of weights without the possibility of attributing to a particular element of the system (neuron, weight, connection . . .) a very precise representational function within this global type of representation.

Genetic Algorithms

The biological metaphor that constitutes the inspiring principles for the development of the genetic algorithms (GAs) is that of Darwinian evolution, based on the principle of the "only the fittest survive" strategy. Individuals compete in nature for food, water refuge, and attracting a partner. The most successful individuals survive and have a relatively large number of offspring. Then their (outstanding) genetic material is transmitted to an increasing number of individuals in each successive generation. The combination of such genes (of such outstanding characteristics) produces individuals whose suitability (fitness) to the environment sometimes transcends that of their parents. In this way, species evolve. John Holland (17) and his colleagues at the University of Michigan are unanimously recognized as the first researchers to envisage utilizing this strategy for solving the usual computer science problems.

Then the first step in utilizing the GA approach consists in creating a population of individuals (from a few tens to a few hundreds) represented by chromosomes (sometimes called genotypes). From the viewpoint of the problem to be solved, each chromosome represents a set (list) of parameters that constitutes a potential solution for the problem. For example, in a problem requiring a numerical solution, a chromosome may represent a string of digits; in a scheduling problem, a chromosome may represent a list of tasks; in a cryptographic problem, a string of letters. Each item of the list is called a "gene." Traditionally, the parameters (genes) are coded by some sort of binary alphabet. For example, let us suppose we are using GAs to optimize a function $f(x, y, z)$. Then a chromosome (a possible solution) consists of three genes (the three variables), each represented in binary form, for example in 10 bits, which means that we have a range of 1024 discrete values that can be associated with each variable. Then a chromosome takes the form of a string of 30 binary digits. Note, however, that this binary technique is not at all mandatory.

The fitness function constitutes another essential aspect of the GA approach. It consists of some predefined criterion of quality that is used to evaluate the utility of a given chromosome (of a solution). Because the fitness of a solution is always defined with respect to the other members of the population, the fitness for a particular chromosome is sometimes defined as f_i/f_{av}, where f_i is the result produced for the chromosome by an evaluative function that measures performance with respect to the chosen set of parameters (genes), and f_{av} is the average result of the evaluation for all of the chromosomes of the current population. In an optimization problem of a function $f(x, y, z)$, like that mentioned previously, the fitness function simply corresponds presumably, to an absolute minimum or maximum of the function but, in other problems, it measures, for example, a number of generations, a processing time, a real cost, a particular parametric ratio.

The fitness function alone would only permit statistically selecting some individuals without improving the initial or current population. This task is achieved by the genetic operators, crossover and mutation. In a given population some individuals are selected for reproducing with a probability (stochastic sampling) proportional to their fitness. Then the number of times an individuals is chosen represents a measure of its performance within the original population. In conformance with the principle of the "strongest survive" paradigm, outstanding individuals have a better chance of generating a progeny, whereas low-fitness individuals are more likely to vanish.

Crossover takes two of the selected individuals, the "parents", and cuts their gene strings at some randomly (at least in principle) chosen position, producing two head and two tail substrings. Then the tail substrings are switched, giving rise to two new individuals called offspring, each of which inherit some genes from each of the parents. The offspring are created through the exchange of genetic material. Crossover is considered the most important genetic operator because it can direct the search towards the most promising regions of the search space. Mutation is applied to the offspring after crossover, and consists of random modification of the genes with a certain probability (normally small, e.g., 0.0001) called the mutation rate. Mutation's function is reintroducing divergence into a converging population, that is, ensuring that no point in the search space is neglected during processing. In fact a correct GA should converge, which means that, generation after generation, the fitness of the average individual must come closer to that of the best individual, and the two must approach a global optimum. Mutation can be conceived, in biological terms, as an error in the reproductive process, the only way to create truly new individuals (crossover uses of already existent genetic material).

Solving a problem using a GA approach consists of developing a sort of biological cycle based on selecting the fittest individuals and using the genetic operators, which can be visualized with the algorithm of Fig. 5.

Genetic algorithms are part of a wider family of biologically inspired methods generally called evolutionary algorithms (18), which are search and optimization procedures all based on the Darwinian evolution paradigm discussed at the beginning of this section. They consist of simulating the evolution of particular individuals by applying the processes of selection, reproduction, and mutation. Apart from GAs, other evolutionary methodologies are known under the name of Evolutionary Strategies, Evolutionary Programming, Classifier Systems, and Genetic Programming. Genetic Programming has emerged in recent years as particularly important. Briefly, Genetic Programming can be seen as a variation of GAs where the evolving individuals are computer programs instead of chromosomes formed of fixed-length bit strings. When executed, the programs solve the given problem (19). One of the main features of Genetic Programming is that the programs are not represented as lines of ordinary code, but rather as parse trees corresponding to a coding syntax in prefix form, analogous to that of the LISP programming language. The nodes of the parse trees correspond to predefined functions (function set) that are supposed to be appropriate for generally solving problems in a domain of interest, and the leaves, that is, the terminal symbols, correspond to the variables and constants (terminal set) that are suited to the problem under consideration. Then crossover is implemented by swapping randomly selected subtrees among programs. Normally, mutation is not implemented.

Now we add some details about crossover, the most impressive of the GA techniques. Figure 6 is an example of single-point crossover. For simplicity, we suppose we are dealing with the optimization of an $f(x)$ function. In this case, the two parent chromosomes represent two values of x, coded as 10-bit binary numbers ranging between 0000000000 and 1111111111. These values represent the lower and upper bounds of the validity interval for x. To operate the crossover, a random position in the chromosome string is selected, six in Fig. 6. Then the tails segments are swapped to produce the offspring, which are then inserted in the new population in place of their parents. Note that crossover is not systematically applied to all the possible pairs formed by the individuals selected for reproduction, but it is activated with a crossover rate typically ranging from 0.6 to 1.0, as compared with the very low mutation rate, see previous discussion. Mutation consists of changing, for example, the second offspring of Fig. 6 from 1110011**0**10 to 1110011**1**10, assuming then that the eighth gene has been mutated (we identify here, for simplicity's sake, bits with genes). After producing a certain number of generations, we should find (for our minimization problem) a set of values of x, corresponding to the best chromosomes in each generation, all clustered around the value of x corresponding to the absolute minimum of $f(x)$. Note that crossover and mutation can produce new chromosomes characterized by fitness lower than the fitness of the parents, but they are unlikely to be selected for reproduction in the next generation.

Single-point crossover, as illustrated in Fig. 6, is not the only technique used to execute crossover. In the two-point crossover, each chromosome to be paired is absorbed into a

```
BEGIN /* genetic algorithm */
      produce an initial population of individuals
      evaluate the fitness of all the initial individuals

      WHILE termination condition not satisfied DO
      BEGIN /* produce a new generation */
            select fitter individuals for offspring production
            recombine the parents' genes to produce new individuals
            mutate some individuals stochastically
            evaluate the fitness of all the new individuals
            generate a new population by inserting some good news
                  individuals and by discarding some old bad ones
      END
END
```

Figure 5. Pseudocode schematically describing a standard genetic algorithm.

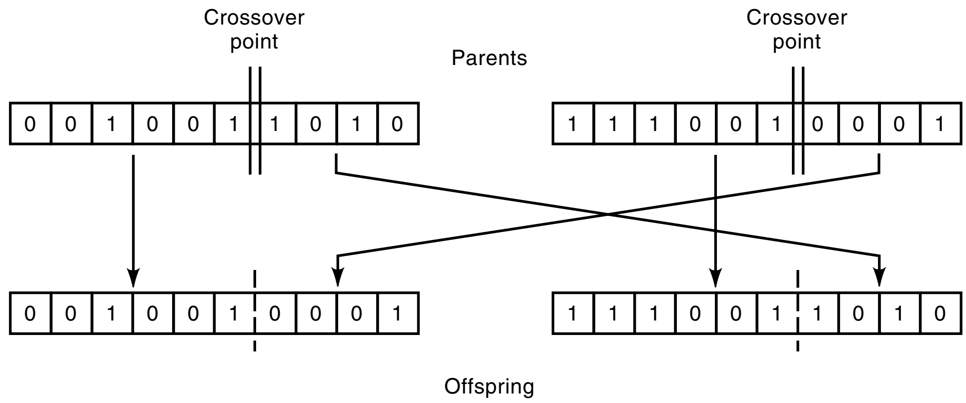

Figure 6. The single-point crossover technique. Two offspring are generated from two parents by a crossover of *length* 4 (the length of the tail segments). The tails segments are swapped to simulate an exchange of genetic material.

ring by joining the ends of the bit string together. To remove a segment from this ring and execute swapping, two cut points are necessary. Multipoint crossover operates according to the same principle. Other techniques exist, for example, uniform crossover (see Refs. 20 and 21).

From the viewpoint of view of symbolic, biologically inspired approaches, it is clear that the GA solution, like the neural network approach examined in the previous section, does not satisfy the requirements of (1) having all the elements of the problem explicitly represented by dedicated symbols and (2) being able to trace exactly the contribution of these symbols in constructing the final solution. As for neural networks, an explicit representation of the intervening factors is given only for the input and output values, and stochastic processes come in at any step of the global procedure (see the previous discussion of the stochastic sampling that selects some individuals for reproduction, where the probabilities are represented by the crossover and mutation rates). There is no accepted general theory which explains exactly why GAs have the properties they do (22, p. 64). One of the first attempts to explain rigorously how GAs work is given by the so-called schema theorem proposed first by Holland (17). A schema is a pattern or template that, according to the usual binary coding option, corresponds to a string of symbols chosen in the following alphabet: {0, 1, #}; # is a wild card symbol that can stand for both 0 and 1. Then schema like [1#0#1] is equivalent to the following family of strings (chromosomes or parts of chromosomes): [10001], [10011], [11001], [11011]. Holland's idea was that, having evaluated the fitness of a specific string, this value could also supply partial information about all of the strings pertaining to the same family. Then the influence of the basic GA operations, selection, crossover and mutation, on the good behavior of an algorithm could be established by evaluating their action on the schemata. By determining the "good" schemata, and by passing these to the chromosomes produced in each following generation, the probability of producing even better solutions could be increased.

Three of the essential parameters that intervene in the schema theorem are the length l of a schema (the global number of symbols, five in the above schema), the defining length δ, and the order o of a schema. δ is the distance between the first and the last non-# symbols in a schema (again five in the previous schema, three for the schema [#0#1]); $o = l -$ number of the # symbols (three for the first schema, two for the second). Now an exponential growth of schemata having a fitness value above the average value in the subsequent generation of a GA can be shown (schema theorem) for low-order schemata with short δ. Because δ is a parameter linked with crossover, and o with mutation, a search for the condition of an optimum behavior of a GA can limit itself to considering δ. A building block is an above average schema with a short δ. Then the power of a GA consists of being able to find good building blocks. Successful coding option is an option that encourages the emerging of building blocks, etc. These results are obtained under very idealized conditions and can only supply very general indications of trend.

The classical reference in the GAs field is Ref. (23). Refs. 22 and 24 are two good introductory papers. Ref. 25 is a more advanced introduction.

Some Remarks About Fuzzy Knowledge Representational Techniques

The fuzzy logic paradigm is also based on some sort of biologically inspired approach, even if the analogy looks less evident. It consists of the fact that fuzzy logic intends to simulate the way humans operate in ordinary life, that is, on a continuum, not according to crisp all-or-nothing Aristotelian logic. Humans use, for example, some forms of gradually evolving linguistic expressions to indicate, with respect to a given thermal environment, that they are comfortable, cold, or freezing. Fuzzy logic allows quantifying such fuzzy concepts representing our sensations about temperature by using numeric values in the range of 0 (e.g., comfortable) to 1 (e.g., freezing and 0.7 representing "cold").

More precisely, according to the fuzzy sets theory every linguistic term expressing degrees of qualitative judgements, like tall, warm, fast, sharp, close to etc., corresponds to a specific fuzzy set. This theory, introduced by Zadeh in Ref. 26, is the core of the fuzzy logic paradigm; see also Refs. 27 and 28. The elements of the set represent different degrees of membership able to supply a numeric measure of the congruence of a given variable (e.g., temperature) with the fuzzy concept represented by the linguistic term.

In very simple terms, knowledge representation according to the fuzzy logic approach consists in computing the degree of membership with respect to a group of fuzzy sets for a collection of input values. For example, we will assume that, for a fuzzy application dealing with a temperature regulating system, the fuzzy sets to be considered for the variable "temperature" are cold, cool, comfortable, warm and hot. The process that allows us to determine, for each of the inputs, the corresponding degree of membership with respect to each one of the defined sets is called "fuzzification." The degrees are calculated by using appropriate membership functions that characterize each one of the sets. The values resulting from the calculus are collected into fuzzy input variables like, for example, *temperature_is_cold.*

The definition of the membership functions for the fuzzy sets is essential for executing the fuzzification process. Usually, the functions are created experimentally on the basis of the intuition or experience of some domain expert. Even if any suitable mathematical function can be used, at least in principle, to represent the membership, normally only triangles and trapezoids are utilized because their use favors all of the operations of construction, maintenance, manipulation. For example, Figure 7 shows some possible membership functions for the five fuzzy sets introduced previously. As can be seen on this figure, an input value of 83°F is translated into two fuzzy values, 0.2 which represents the degree of membership with respect to the fuzzy set "hot," and 0.8 representing the degree of membership with respect to the fuzzy set "warm." Imprecise, approximate concepts like warm and hot are translated into computationally effective, smooth, and continuous terms.

Then the fuzzy values calculated by using the membership functions are utilized within systems of if-then rules in the style of "If the temperature is warm and the humidity is high, then cooling must be maximum and fans speed is high." In a rule like this, humidity, cooling, and speed are obviously, like temperature, defined in terms of fuzzy sets and associated membership functions. There will be, for example, a triangle- or trapezoid-shaped function that represents the membership function for the fuzzy set "speed equals high." The actual values of the variables, like temperature and humidity mentioned in the antecedents (if parts) of the rules are translated into the corresponding fuzzy values (degrees of membership) computed through the fuzzification process. Then a truth value for the rule can be calculated. Normally, it is assumed that this corresponds to the weakest (last-true) antecedent fuzzy value, but other methods can be used, for example, multiplying all of the fuzzy values of the antecedent together. The truth value obtained is associated with all of the fuzzy sets representing the variables, like cooling and speed, that make up the consequent (then part) of the rule. Then we could find, for example, that the degree of membership (fuzzy output) for the fuzzy set "speed equals high" is 0.8. Fuzzy outputs must then be "defuzzificated" to obtain crisp values, for example, an exact value for the speed of the fans. A common defuzzification technique is the centroid (or center-of-gravity) method; see Ref. 29 for the technical details.

As a last remark, we can note that, when a fuzzy logic system is in operation, the membership functions are fixed. However, it is possible to envisage fuzzy systems that employ adaptive techniques to adjust their membership functions and are therefore better able to better reflect a given environment. It is also possible to use adaptive techniques to dispose of an evolving system of rules. In this case, a close relationship with neural network systems can then be established. For a recent paper on the theory of fuzzy neural integration see Ref. 30. See Ref. 31 for an example of a neuro-fuzzy learning control system.

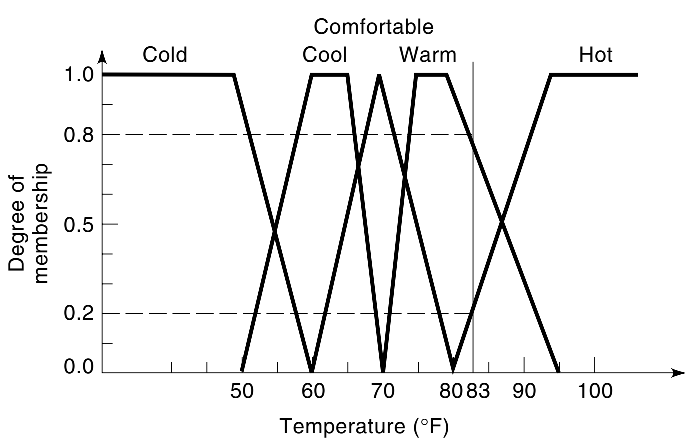

Figure 7. Membership functions for the five fuzzy sets cold, cool, comfortable, warm, and hot defined for the variable temperature. An input value of 83°F is translated (knowledge representation) into two fuzzy values, 0.2 (degree of membership with respect to the fuzzy set hot), and 0.8 (degree of membership with respect to the fuzzy set warm).

THE SYMBOLIC APPROACH

A symbol is a physical mark that can be reproduced and that can be associated with a precise and unequivocal meaning by an observer. According to A. Newell and H. A. Simon (32), "Physical symbol systems are collections of patterns and processes, the latter being capable of producing, destroying and modifying the former" (32, p. 125). In practice, the knowledge representational paradigms associated with the symbolic approach range between two possible basic forms:

- Pure rule-based representations supporting inference by resolution. Inside this first pole, we can differentiate the systems developed in a logic programming context from the simplest Expert Systems shells based on the production rules paradigm.
- Pure frame- or object-based representations, supporting inference by inheritance, defaults and procedural attachment. A particular class of inheritance-based systems that are particularly fashionable today are the so-called description logics (or terminological logics) systems.

In the following, we deal first with the resolution principle and its associated representational systems, logic programming, and production rules. Then we describe the inheritance principle, and the corresponding representational systems, frames—more generally knowledge engineering software environments (KESEs)—and the terminological languages. We do not deal explicitly with a (once very popular) knowledge representational paradigm like semantic networks because the modern realization of this paradigm coincides practically with the frame-based systems. See, however, Ref. 33. For advanced types of representation which are derived in some way from semantic networks, like Conceptual Graphs and Narrative Knowledge Representation Language (NKRL), see, respectively, Refs. 34 and 35.

The Resolution Principle

The resolution principle originates in the area of automatic theorem proving, an AI discipline where people try to use computers to prove that a theorem, that is, a clause (see later) whose truth value is yet unknown, can be derived from a set of axioms, that is, clauses that are assumed to be true. The resolution principle was introduced by J. A. Robinson in a famous paper (36); see also Ref. 37.

In its most simple formulation (chain rule), the resolution principle can be reduced to an inference rule expressed as

$$\text{From } (A \lor B) \text{ and } (\neg A \lor C), \text{ deduce that } (B \lor C) \qquad (3)$$

In Eq. (3), we follow the usual conventions of the predicate calculus in logic. Then A, B, and C are atomic formulas or literals, that is, in their most general form they are expressions of the type $P(t_1 \ldots t_n)$ where P is a predicate and $t_1 \ldots t_n$ are terms. Predicates represent statements about individuals, both by themselves and in relation to other individuals. From a semantic point of view, they can assume the value of either TRUE or FALSE. Terms may be constant symbols, like "Peter." Constant symbols are the simplest form of term, and they are used to denote the interesting entities (physical objects, people, concepts, etc.) in a given domain of discourse. Then a simple atomic formula can be "love (Peter, Mary)" where "love" is the predicate, and "Peter" and "Mary" are the terms. But terms may also be variables or expressions of the form $f(t_1 \ldots t_n)$, where f is a n-place function and $t_1 \ldots t_n$ again are terms. It is important to recall here that functions, as the difference of predicates, do not return TRUE or FALSE, but that they behave like operators returning objects related to their arguments. For example, the function "father-of" applied to the argument (a term represented by a constant symbol) "Peter" would supply the value "John." The symbols $\lor$ and $\neg$ (logical connectives) represent, respectively, the inclusive or and the negation.

The "disjunctions" $(A \lor B)$, $(\neg A \lor C)$ and $(B \lor C)$ in Eq. (3) are particularly important types of well-formed formulas (wff) of the first-order predicate calculus called clauses. It can be shown (see also later) that each standard expression of the predicate logic can be reduced to a set of disjunctive clauses, where the variables possibly included in the clauses are (implicitly) universally quantified. However, the intuitive meaning (the direct translation into an English statement) of the original logic expression is often completely lost after the translation into clausal form.

From Eq. (3), it is evident that the resolution process, when applicable, can take a pair of parent well-formed formulas (wffs) in the form of clauses to produce a new, derived clause (the resolvent), on condition that one of these clauses contains a literal (atomic formula), $\neg$ A, which is the exact negation of one of the literals, A, in the other clause. The literals A and $\neg$ A appear as cancelled. Then the resolution method for automatic theorem proving is a form of proof by contradiction. In its more general formulation this method consists in assuming that, if a theorem follows from its axioms, the axioms and the negation of the theorem cannot be simultaneously true. The proof of a theorem using resolution is as follows:

1. Negate the theorem to be proved, and add the negated theorem to the list of axioms.
2. Put the new list of axiom in clausal form, obtaining then a global set of clauses.
3. Simplify the clauses and produce the corresponding resolvents through the application of the chain rule Eq. (3) to the clauses of the global set.
4. Add these resolvents to the global set, and recursively produce new resolvents through the systematic application of Eq. (3).
5. Halt the procedure when a contradiction can be found, that is, when an empty clause is produced. In this case, report that the theorem is TRUE. If the empty clause, sometimes noted as $\square$, cannot be produced, report that the theorem is FALSE.

Resolution is a particularly powerful procedure because it can be shown that resolution is complete for first-order predicate logic (i.e., it can prove all of the theorems in this particularly useful form of logic). Moreover, it is sound, that is, it will not affirm that some nontheorems are true. Note, however, that, if the theorem is FALSE (i.e., the empty clause cannot be produced), the process generally cannot terminate.

As a very simple example, let us consider the case of the well-known modus ponens in logic, which affirms that, from p and $p \supset q$ (i.e., p and $p \supset q$, the axioms, both have a truth value = TRUE), we can deduce the theorem q. Using the logical equivalence: $\neg x_1 \lor x_2$ eq. $x_1 \supset x_2$, we can reduce the two axioms to the clauses (1) p and (2) $\neg p \lor q$, which are congruent with Eq. (3). Now we have to add to these two clauses a third clause given by the negation of the theorem, that is, (3) $\neg q$. Resolving the three clauses against each other leads immediately to the reciprocal cancellation of p and $\neg p$ in (1) and (2), leaving us with the final contradiction q and $\neg q$. Then the theorem has the truth value TRUE.

From what we have said until now, a first phase in the resolution process consists of converting the (negation of the) theorem and the axioms into a set of disjointive clauses. Even if, as already stated, it can be proven that this conversion is feasible for any possible wff, the real implementation can be relatively complex, especially in the presence of functions, variables and quantifiers. The details of this conversion process can be found in (Ref. 38, pp. 145–149). It consists of a series of transformations that use well-known properties of the predicate calculus and result in eliminating the symbols different from $\lor$ and $\neg$ and the quantifiers $\forall$ (for all) and $\exists$ (there exists) in a progressive simplification of the original formulas.

For example, the first step of the transformation process consists of getting rid of the implication symbol, $\supset$. This is eliminated by using the property: $x_1 \lor x_2$ eq. to $\neg x_1 \supset x_2$. The two de Morgan laws: $\neg (x_1 \land x_2)$ eq. to $\neg x_1 \lor \neg x_2$; $\neg (x_1 \lor x_2)$ eq. to $\neg x_1 \land \neg x_2$ are used to reduce the scope of the negation symbols, that is, to constrain the negation symbols to apply to at most a single literal (moving inward). Existential quantifiers $\exists$ are generally simply eliminated by introducing a constant c, for example, $\exists\ x\ P(x)$ is replaced by $P(c)$. Then we claim that an x exists by selecting a particular constant to replace x. Existential quantifiers $\exists$ that occur within the scope of a universal quantifier $\forall$ present additional problems. They are eliminated by replacing their variables with a function (skolem function) of the universally quantified variable. Let us consider, for example, $\forall y\ \exists x\ P(x, y)$, to be read as "for all y, there exists an x such that $P(x, y)$." Because the existential quantifier is within the scope of the universal quantifier, we can suppose that the x "that exists" depends on the value of y, that is, that it is always possible to find a function that takes argument y and systematically returns a proper x. A function like this is called a skolem function, Skolem(y), which maps each value of y into x. Then using this Skolem function in place of the x "that exists," we can eliminate the existential quantifier and rewrite the original formula as $\forall y$ P[(Skolem(y), y)]; (see Ref. 38, pp. 146–47). The explicit occurrences of the symbol $\wedge$, "and," in the transformed formula are eliminated, with breaking this formula into a set of disjointed clauses as required by the resolution principle. This makes sense because each part of a conjunction must be TRUE for the whole conjunction to be TRUE. The transformation process also includes (1) renaming, if necessary, all of the (universally quantified) variables so that no two variables are the same in different disjunctive clauses; (2) eliminating the universal quantifiers (in reality, this elimination is only a formal step because, as already stated, all the variables are assumed implicitly universally quantified within the resulting clauses).

Now we must mention a very important point about the resolution principle. As we have seen before, a fundamental step in the procedure consists in identifying two literals, A and $\neg$ A, where the second is the exact negation of the first. This allows us to eliminate the two. If the literals are reduced to atomic constants or if the terms they include do not imply the presence of variables, their identification is immediate. This is not true when variables and skolem functions are present. To give a simple example, to cancel the literals P(a) and $\neg$ P(x), where a is a constant and x a variable, it is necessary to recognize that (1) the literal $\neg$ P(x) asserts that there exists no x for which P(x) is true (x is universally quantified), while (2) P(a) asserts that there is an object a for which P(a) is true. Then generally to be authorized to cancel two literals, it is necessary first to execute their "unification." We recall here that unification is informally defined as the process of finding a common substitution instance for the arguments of the predicates making up two literals that render these literals identical; see Ref. 39. In our case, the substitution instance is obviously the constant a.

Logic Programming

Logic programming refers to a programming style based on writing programs as sets of assertions in predicate logic (clauses): these clauses have both (1) a declarative meaning as descriptive statements about entities and relations proper to a given domain (knowledge representation) and, in addition, (2) they derive a procedural meaning because they are executable by an interpreter. This last process is based solely on the resolution principle, where unification involving a pattern matching algorithm represents the central element. Restriction to a resolution theorem prover for the Horn clauses subset of logic (see Ref. 40 and 41) provides the logical basis for the well-known programming language PROLOG (PROgramming in LOGic), and supplies PROLOG and its derivatives with a relative tractability of deductions; see also later.

As we have already seen, a clause is a particular form of logical formula that consists of a disjunction of literals, that is, a disjunction of atomic formulas and of negations of atomic formulas. Then we can generally write a clause as

$$A_1 \vee A_2 \ldots \vee A_m \vee \neg B_1 \vee \neg B_2 \ldots \vee \neg B_n \quad m, n \geq 0 \tag{4}$$

Now clause (4) can be written as $A_1 \vee A_2 \ldots \vee A_m \vee \neg (B_1 \wedge B_2 \ldots \wedge B_n)$ using one of the two of de Morgan's laws, and then: $\neg (A_1 \vee A_2 \ldots \vee A_m) \supset \neg (B, B_2 \ldots \wedge B_n)$ using the equivalence: $x_1 \vee x_2$ eq. to $\neg x_1 \supset x_2$. Now we can use the so-called contrapositive law: $x_1 \supset x_2$ eq. to $\neg x_2 \supset \neg x_1$ (see Ref. 38, p. 138) to write Eq. (4) as:

$$(B_1 \wedge B_2 \ldots \wedge B_n) \supset (A_1 \vee A_2 \ldots \vee A_m) \tag{5}$$

The result obtained is particularly interesting because it affirms that any clause is equivalent to an implication, where $(B_1 \wedge B_2 \ldots \wedge B_n)$ is the antecedent, or the conditions of the implication, and $(A_1 \vee A_2 \ldots \vee A_m)$ is the consequent, or the conclusion of the implication. Stated in different terms, Eq. (5) says that, if the different conditions $B_1, B_2 \ldots, B_n$ are all verified (TRUE), they imply a set of alternative conclusions expressed by $A_1, A_2, \ldots, A_m$. The standard conventions for expressing implications, (see Ref. 42, pp. 425–427), elude the use of the usual logical connectives like $\wedge$, $\supset$ and $\vee$. Then we write Eq (5) as

$$A_1, A_2, \ldots, A_m \leftarrow B_1, B_2, \ldots, B_n \quad m, n \geq 0 \tag{6}$$

where the arrow $\leftarrow$ is the connective "if" that represents the implication, $B_1, \ldots, B_n$ are the *joint* conditions and $A_1, \ldots, A_m$ the *alternative* conclusions. $B_1, \ldots, B_n$ and $A_1, \ldots, A_m$ are literals (atomic formulas), as defined previously. We can add that the variables $x_1, \ldots, x_k$ that can appear in a clause C are implicitly governed by the universal quantifier $\forall$, so that a clause C like the clause represented by Eq. (6) is, in reality, an abbreviation for $\forall x_1, \ldots, \forall x_k$ C. Examples of clauses written according to the format of Eq. (6) are: "Grandparent(x, y) $\leftarrow$ Parent(x, z), Parent (z, x); (m = 1)," which expresses the implication that "x is grandparent of y if x is parent of z *and* z is parent of y", and "Male(x), Female(x) $\leftarrow$ Parent(x, y);(n = 1)," saying that "x is male *or* x is female if x is parent of y," where the alternative or/and is linked with the different meaning of the symbol "," in the condition ("and") and conclusion ("or") segments of the implication; see the original formula in Eq. (5).

Horn Clauses. Now we can introduce the Horn clauses (named after Alfred Horn, who first investigated their properties). Horn clauses are characterized by having at most one positive literal. Then expression (4) can be written as

$$A \vee \neg B_1 \vee \neg B_2 \ldots \vee \neg B_n \quad n \geq 0 \tag{7}$$

Executing the same transformations on (7) we have applied to (4) and expressing the result according to the standard convention, we obtain finally:

$$A \leftarrow B_1, B_2, \ldots, B_n \quad n \geq 0 \tag{8}$$

Eq. (8) translates the fact that Horn clauses represent a particular sort of implication which contains at most *one* conclusion. Restriction to Horn clauses is conceptually equivalent to disallowing the presence of disjunctions ($\vee$) in the conclusive part of the clause. Note that, in Eq. (8), we can now give to the comma, ",", the usual meaning of "logical and", $\wedge$. When $n = 0$, the implication becomes an assertion, and the symbol $\leftarrow$ can be dropped. Then the following example: Grandparent(John, Lucy), asserts the fact that John is a grandparent of Lucy. The interest in using Horn clauses, less expressive, from a knowledge representational point of view, than the general clauses considered until now, is linked with the well-known principle (see Ref. 43 and later, the section on terminological logics) that suggests reducing the power of the knowledge representational languages so that formalizing interesting applications is still possible but, at the same time, the corresponding computational tasks are computationally feasible, that is, polynomially tractable or at least decidable. For example, linear algorithms exist for dealing with propositional logic in Horn clauses form (see Ref. 44).

Until now, we have implicitly associated a declarative meaning with our (Horn) clauses, which represent then static chunks of knowledge such as x is grandparent of y if x is parent of z and z is parent of y (whatever may be the values of the variables x and y) or John is a grandparent of Lucy. But we can also associate a procedural meaning with a clause like Eq. (8). In this case, and assuming a top-down resolution strategy, Eq. (8) may be viewed as a procedural declaration that reduces the problem of the form A to subproblems B_1, B_2, . . ., B_n, where each subproblem is interpreted in turn as a procedural call to other implications. The conclusion A of the implication is the head or the name of the procedure, and it identifies the form of the problems that the procedure can solve. The procedural calls B_i, or goals, form the body of the procedure. Looked at this way, the first example previously (an implication) can be interpreted as follows: to find an x that is a grandparent of y, try to find a z who has x as a parent and who is, in turn, a parent of y, and the second (an assertion) can be interpreted as follows: when looking for the grandparent of Lucy, return the solution John.

Now to complete the procedural interpretation of Horn clauses and to show how this interpretation is perfectly coherent with the mechanisms of the resolution principle introduced in the previous section, we must introduce, after the "implications" and the "assertions," a third form of Horn clause, the "denials." In this case, the literal A of Eq. (8) disappears, and a denial is represented as $\leftarrow B_1, B_2, \ldots, B_n$, with $n > 0$. The name "denial" comes from the fact that, if we drop the only positive literal A from the original expression of a Horn clause Eq. (7), and we apply one of the two of de Morgan's law, Eq. (7) is transformed into: $(\neg B_1 \vee \neg B_2 \ldots \vee \neg B_n)$ eq. $\neg (B_1 \wedge B_2 \ldots \wedge B_n)$. Then, a denial like: $\leftarrow$ Male (x), Grandparent(x, Lucy), means literally, in a declarative interpretation, that, for no x, x is male and he is the grandparent of Lucy.

Denials are used in a logic programming context to express the problems to be solved. To be congruent with the resolution principle process, we assume that a particular denial (all the denials comply with the clause format) is the negation of the theorem to be proved and, as usual, we will add the denial to the existing assertions and implications (clauses), the axioms, to try to obtain the empty clause, therefore proving the theorem. Returning to the previous example, Male(x), Grandparent(x, Lucy), this can represent a theorem to be proved. In the procedural interpretation, we will assume this as query that, according to the top-down strategy chosen (see above), characterizes the starting point of the normal resolution process. Unification must, of course, be used to derive the empty clause □ that, according to the procedural interpretation, now can be considered a STOP instruction.

Following Ref. 42 (p. 428), now we can describe the general format of a logic program (slightly) more formally. Let us assume a set of axioms represented by a set of Horn clauses (8), and let us assume the procedural interpretation. The conclusions we can derive from the previous set must, according to the resolution principle, be negated (i.e., represented as a denial) and added to the set of axioms. According to what is already expounded, they are expressed as a clause of the form Eq. (9), consisting solely, according to the procedural interpretation, of procedural calls C_i which behave as goals:

$$\leftarrow C_1,\ C_2,\ \ldots,\ C_m \quad m > 0 \tag{9}$$

Now the proof consists of trying to obtain the empty clause □ through a resolution process, expressed as follows. A procedural call C_i in the goal statement Eq. (9) invokes a procedure Eq. (8) pertaining to the original set of axioms according to the following modalities:

a. by unifying the call C_i in Eq. (9) with the head (the name) of Eq. (8);
b. by replacing the call C_i in Eq. (9) with the body of Eq. (8), then the new goal statement is

$$\leftarrow C_1, \ldots, C_{i-1}, B_1, \ldots, B_n, C_{i+1}, \ldots, C_m;$$

c. by applying the substitution instance σ to Eq. (9),

$$\leftarrow (C_1, \ldots, C_{i-1}, B_1, \ldots, B_n, C_{i+1}, \ldots, C_m)\sigma,$$

where σ replaces variables by terms to render the head A and the call C_i identical, $A\sigma = C_i\sigma$.

Now we give a very simple, self-evident example. Let us suppose the following set of Horn clauses, which includes both implications and assertions:

1. Grandparent(x, y) $\leftarrow$ Parent(x, z), Parent (z, y)
2. Parent(x, y) $\leftarrow$ Mother(x, y)
3. Parent(x, y) $\leftarrow$ Father(x, y)
4. Father (John, Bill)
5. Father (Bill, Lucy).

Note that 2 and 3 are the Horn equivalents of a general implication which could be expressed as follows: Father(x, y), Mother(x, y) $\leftarrow$ Parent(x, y), that is, "x is the father of y *or* x is the mother of y if x is parent of y." Now we will use a goal statement like

6. $\leftarrow$ Grandparent(John, Lucy)

that is we want to prove that John is really a grandparent of Lucy. According to the previous algorithm, we must find (a), a clause head which can unify the (unique) procedural call given by 6. This clause head is, of course, the head of 1, and the unification produces, see (c), the bindings x = John, y = Lucy. Taking these bindings into account and applying step

(b) of the algorithm, we obtain a new goal statement from the body of 1:

7. ← Parent(John, z), Parent (z, Lucy).

Again we apply the algorithm using the first procedural call C_1 of 7, that is, Parent(John, z). This unifies both the heads of 2 and 3 producing two new goal statements, 8 and 9, with the bindings x = John, $y = z$:

8. ← Mother(John, z), Parent (z, Lucy)
9. ← Father(John, z), Parent (z, Lucy).

The procedural call C_1 of 8, Mother(John, z), fails to unify the set of Horn clauses. The procedural call C_1 of 9, Father(John, z), on the contrary unifies with 4 linking z to Bill. Given that 4 is not endowed with a body, the steps (b) and (c) of the algorithm simply reduce the goal statement 9 to Parent(Bill, Lucy) that, through 3, becomes Father(Bill, Lucy) finally producing the empty clause □ through the unification with 5.

PROLOG AND DATALOG. Now, if we substitute the symbol "←" in Eq. (8) with ":-", with the same meaning, we obtain the usual representation of a PROLOG clause:

$$\mathrm{A} : -\mathrm{B}_1,\ \mathrm{B}_2,\ \ldots,\ \mathrm{B}_n \quad n \geq 0 \tag{10}$$

where A (the head) and B_i (the body) have the same interpretation as in the previous sections and the symbol ":-" stands for the logical implication "from right to left", meaning that, to solve the goal expressed in the head, one must solve all subgoals expressed in the body. A fact is represented in PROLOG by a headed clause with an empty body and constant terms as the head's arguments: father(Bill, Lucy). A rule is represented by a headed clause with a nonnull body. See the well-known PROLOG example

$$\text{ancestor}(X, Y)\text{ :- father}(Z, Y), \text{ancestor}(X, Z)$$

which means that, for all of the PROLOG variables X, Y, and Z, if Z is the father of Y and X an ancestor of Z, then X is an ancestor of Y. A query is represented by a headless clause with a nonempty body, for example, :-father(Lucy), "who is the father of Lucy?." A query without variable arguments produces a "yes" or "no" answer. See -father(Bill, Lucy), "is it true that Bill is the father of Lucy?." PROLOG was originally a strongly constrained resolution theorem prover. About 1972, it was turned into a normal programming language to implement a natural language question-answering system by a team led by Alain Colmeraurer in Marseilles; see Refs. 45, 46. Then van Emden and Kowalski (47) provided an elegant formal model of the language based on Horn clauses.

To fulfill its functions as a normal programming language, PROLOG introduces, however, several important modifications (some extralogical features) with respect to the pure logic programming paradigm. First, it must obviously introduce some built-in predicates for input and output to allow clauses to be read and written to and from terminals and databases. Secondly, PROLOG adopts a very strict discipline for control. When executing a program, that is, when seeking the match of a literal in the goal statement (query) against the head of some clause and then to substitute the goals (if any) in the body of that clause for the original literal in the query (see the logic programming example illustrated before), PROLOG follows these two rules:

- The clauses that together make up the program are tested strictly in the order they appear in the text of the program. In the current goal statement, the leftmost literal (procedural call) is systematically chosen.
- When a success or a failure is attained, the systems backtracks, that is, the last extensions (substitutions, transformations) in the goal statement are undone, the previous configuration of the statement is restored (chronological backtracking), and the system looks for alternative solutions starting from the next matching clause for the leftmost literal of the reinstated statement.

In practice this means, among other things, that PROLOG's goals are executed in the very order in which they are specified. Therefore, PROLOG programmers order their goals so that the more selective ones are declared first. To optimize this search mechanism (i.e., depth-first search with backtracking), PROLOG uses other extralogical features, like the built-in predicates "fail" (which automatically triggers a failure) and "cut". Cut is represented as "/" or "!" and it is used to limit searches in the choice-tree which are too expensive because of the systematic use of backtracking (see later). Moreover, PROLOG provides some limited data structures (e.g., lists, trees), means for dealing with variables (e.g., isvar, rreal, integer), and arithmetic. Finally, some utilities for debugging and tracing programs are also provided. Some of these features could also be expressed in first-order logic. Others (read/write, cut) have no logical equivalent.

We will not dwell on the technicalities of the PROLOG programming, which are outside the scope of this article (see, e.g., AI LANGUAGES AND PROCESSING), and we only mention two particularities of this language that have generated a large theoretical debate, that is, the absence of the "occur test" in the standard implementations of PROLOG and the "cut."

As already seen for the resolution method in general and for logic programming in particular, PROLOG makes uses unification extensively. The first modern algorithm for unification proposed by Robinson (36) already contained what is now known as the "occur check." Very informally, it says that, when one of the two terms t_1 and t_2 to be unified is a variable x and when the same variable occurs anywhere in the second term t, that is, if occur (x, t) is true, then the unification fails; see Ref. 39 for more details. The reason for introducing the check is linked with the aim of avoiding any infinite loop because, when trying to unify x and $f(x)$, the substitution σ that renders the two terms identical is $\{x \leftarrow f(f(f(. \ . \ .)))\}$. In the original implementation of PROLOG, Colmerauer left out the occur check for efficiency, e.g., it can be shown, see Ref. 48, that the concatenation of two lists, a linear-time operation in the absence of the occur check, becomes an $O(n^2)$ time operation in the presence of this check. Then PROLOG implementations that follow Colmeraurer are based, more than on unification, on "infinite unification," which can lead in particular cases, to incorrect conclusions.

The cut mechanism allows a programmer to tell PROLOG that some choices made during the examination of the goal chain need not be considered again when the system backtracks through the chain of the goals already satisfied. The main reason for using this mechanism is linked with the fact that the system will not waste time while attempting to satisfy goals that the programmer knows will never contribute to finding a solution. From a syntactical point of view, a cut is equivalent to a goal that is represented by the predicate "!" (or an equivalent symbol) without any argument. Then it can be inserted into the subgoal chain that makes up the right-hand side of a PROLOG clause. As a goal, it is immediately satisfied, and the program continues exploring the chain of goals at its right; as a side effect, it freezes all of the decisions made previously since the clause considered was entered. In practice, this means that all of the alternatives still opened between the invocation of the rule by the parent goal and the goal represented by the cut are discarded.

Now if we transform clause (10) into (11) by adding a cut goal,

$$A: -B_1, B_2, B_3, !, B_4, B_5, \ldots, B_n \quad n \geq 0 \tag{11}$$

the result is that the system backtracks regularly among the three subgoals B_1, B_2, B_3 and, when B_3 succeeds, it crosses the "fence" (the "one-way door") represented by the cut goal to reach B_4 and continues in the usual way, backtracking included, until B_n; see Ref. 49, pp. 66–67. But, if backtracking occurs and if B_4 fails—then causing the fence to be crossed to the left—given that the alternatives still opened have been discarded, no attempt can be made to satisfy goal B_3 again. The final effect is that the entire conjunction of subgoals fails and the goal A also fails.

Apart from its appearance as a "patch" from a strictly logical point of view, the use of the cut introduces some very practical problems, all linked fundamentally with the necessity of knowing perfectly well the behavior of the rules (PROLOG clauses) where the cut must be inserted. In fact given that its use precludes in practice the production of some possible solution, the use of the cut in an environment not completely controlled can lead to the impossibility of producing a perfectly legal solution; again see (Ref. 49, pp. 76–78). To control an expensive tree search, several researchers have suggested using tools external (metalevel control) to the specific clause processing mechanism of PROLOG; see, among many others, the work described in Ref. 50.

In the context of an article about knowledge management, the DATALOG language must be mentioned. It has been specifically designed to interact with large (traditional) databases (DBs) because of the possibility of immediately translating DATALOG programs in terms of (positive) relational algebraic expressions. Its importance in the context of the setup of effective strategies for managing large knowledge bases—at least those conceived under the form of the association of an artificial intelligence component with a (traditional) database management system, see later—therefore is absolutely evident.

From a syntactical point of view, DATALOG can be considered a very restricted subset of general logic programming. In its formalism, both facts and rules are represented as Horn clauses having the general form reproduced in Eq. (12):

$$A: -B_1, B_2, \ldots, B_n \quad n \geq 0 \tag{12}$$

According to the procedural interpretation of Horn clauses Eq. (12) also represents a DATALOG rule, reduced to an assertion or a fact when Eq. (12) consists only of the head A. Then each A or B_i is a literal of the form $P(t_1 \ldots t_n)$ where P is a predicate and t_i are the terms. The basic DATALOG restricts however the type of terms, which can be only *constants or variables,* to the exclusion then, for example, of the *function symbols.* Extension to the basic DATALOG language intended to deal with functions, with the negation of predicates P_i, etc. has been proposed; see also the AI LANGUAGES AND PROCESSING. A literal, clause, rule, or fact which does not contain any variable is called "ground." In particular, to have a *finite* set of all the facts that can be derived from a DATALOG program P, the following two conditions must be satisfied:

- each fact associated with P must be "ground;"
- each variable that appears in the head of a rule of P must also appear in the body of the same rule.

A DATALOG program is a finite set of clauses divided into two disjoint subsets, a set of ground facts, called the extensional database (EDB) and a set of DATALOG rules, called the intensional database (IDB). The important point here is that, given the restriction to constants c_i of the terms included in a DATALOG ground fact, the EDB can physically coincide with a normal, relational database. Now if we call EDB predicates all of those that occur in the EDB and IDB predicates those that occur in IDB without also occurring in EDB, we require as additional conditions that (1) the head predicates of each clause (rule) in IDB (the "core" of the DATALOG program) be only IDB predicates (sometimes, IDB predicates are therefore called intensional predicates) and that (2) EDB predicates may occur in the IDB rules, but only in the B_i (clause bodies). The correspondence between EDB (ground facts) and the relational database is implemented so that each EDB predicate G_i corresponds to one and only one relation R_j of the base. Then each ground fact $G_i(c_1 \ldots c_n)$ of EDB is stored as a tuple $\langle c_1 \ldots c_m \rangle$ of R_j. Also the IDB predicates can be identified with relations, called IDB relations which, in this case, are not stored explicitly in the DB. Therefore they are sometimes called derived or intensional relations and correspond to the "views" of the relational DB theory. The main task of a DATALOG compiler or interpreter is precisely that of calculating these views efficiently. The output of a successful DATALOG program is a relation for each IDB predicate.

Without entering into any further technical details, we can say that

- A DATALOG program P can be considered a query against the extensional database EDB of the ground facts. Then the definition of the correct answer to P can be reduced to the derivation of the least model of P.
- As already stated, a relationship exists between DATALOG and relational databases. Now we can add that DATALOG can deal with recursivity, which is not allowed in relational algebra. On the contrary, relational queries that make use of the "difference" operator cannot be expressed in pure DATALOG. To do this, it is necessary to enrich DATALOG with the logical negation ($\neg$).

We can conclude by saying that DATALOG, as a restricted subset of general logic programming, is also a subset of PROLOG. Hence, each set of DATALOG clauses could be parsed and executed by a PROLOG interpreter. However, DATALOG and PROLOG differ in their semantics. As we have seen, DATALOG has a purely declarative semantics with a strong flavor of set theory. Therefore, the result of a DATALOG program is independent from the order of the clauses in the program. On the contrary, the meaning of PROLOG programs is defined by an operational semantics, that is by the specification of how the programs must be executed. A PROLOG program is executed according to a depth-first search strategy with backtracking. Moreover, PROLOG uses several special predicates, like the cut, that accentuate its procedural character. This strategy does not guarantee the termination of recursive PROLOG programs.

Notwithstanding its nice formal properties linked with its clean declarative style, sometimes DATALOG has been severely criticized from a strictly programming point of view. As a programming language, DATALOG can be considered little more than a toy language, a pure computational paradigm which does not support many ordinary, useful programming tools like those extralogic added to PROLOG to avoid the same sort of criticism. Moreover, from an AI point of view, a very strict declarative style may be dangerous when it is necessary to take control on inference processing by stating the order and method of execution of rules, as happens in many expert systems (ES) shells.

Production Rules as a Knowledge Representational Paradigm

Now returning to formula Eq. (5) given at the beginning of the "Logic Programming" section,

$$(\mathrm{B}_1 \wedge \mathrm{B}_2 \ldots \wedge \mathrm{B}_n) \supset (\mathrm{A}_1 \vee \mathrm{A}_2 \ldots \vee \mathrm{A}_m) \tag{5a}$$

we have already noticed that this formula establishes a very important result, namely, that any clause of first-order logic is equivalent to an "implication," where $(\mathrm{B}_1 \wedge \mathrm{B}_2 \ . \ . \ . \ \mathrm{B}_n)$ is the antecedent or the conditions of the implication, and $(\mathrm{A}_1 \vee \mathrm{A}_2 \ . \ . \ . \vee \mathrm{A}_m)$ is the consequent, or the conclusion of the implication. Formula (5) states that, if the different conditions B_1, B_2, . . ., B_n are all verified (TRUE), they imply a set of alternative conclusions which are expressed by A_1, A_2, . . ., A_m. Expressing (5) succinctly as

$$\mathit{If}\ \mathrm{B}\ \mathit{Then}\ \mathrm{A} \tag{13}$$

where we preserve for B and A the meaning of, respectively, a conjunction and a disjunction of terms, we obtain the well-known notation used for the production rules that still constitutes the basic knowledge representational tool used in a majority of expert systems. Production rules were first introduced in symbolic logic by Emil Post (51) as a general symbolic manipulation system, which used grammar-like rules to specify string replacement operations. An example of such a rule could be $\mathrm{C}_1 X \mathrm{C}_2 \rightarrow \mathrm{C}_1 Y \mathrm{C}_2$, meaning that any occurrence of string X in the contest of C_1 and C_2 would be replaced by the string Y. Then production rules were used in mathematics under the form of Markov normal algorithms (52) and by Chomsky as rewrite rules in the context of natural language processing (53). They became very popular in the AI milieus in the mid-sixties because of the development of the first expert systems, like DENDRAL and MYCIN.

Because of the equivalence between Eqs. (5) and (13), now it is evident that production rules can be interpreted as logical expressions and then submitted to the usual procedures of first-order logic. Also the procedural interpretation that is characteristic of the use of production rules, (see the purpose of the Post's productions mentioned before) is not really contradictory with the basic declarative nature of logic, as appears clearly from the procedural interpretation of Horn clauses. This explains why, whenever it is necessary to establish some theoretically sound result in a particular field involving the application of production rules, the usual strategy consists of converting the set of rules into a set of logic formulas in the form of (5) and then operating on it by using the customary logic tools. As an example, we can mention the recent Vermesan paper (54) where, in the first part, the author explains how a knowledge base of production rules of the form $\mathrm{B}_1 \wedge \mathrm{B}_2 \ . \ . \ . \wedge \mathrm{B}_n \rightarrow \mathrm{A}$ ("$\rightarrow$" is the implication symbol, and B_i and A are first-order literals) can be converted into a set of first-order formulas which are used to set up a theoretical framework to verify the consistency and completeness of the original knowledge base.

Putting Production Systems to Work. A typical system (an expert system) that uses production rules operates in the following way:

- The system contains a rule base, an unordered collection of production rules. In this base, rules r can assume the general form $\mathrm{c}_1 \wedge \mathrm{c}_2 \ . \ . \ . \wedge \mathrm{c}_n \rightarrow \mathrm{a}_1 \wedge \mathrm{a}_2 \ . \ . \ . \wedge \mathrm{a}_m$. This last form does not contradict Eq. (5), as can be seen if we split (5) into as many rules as the terms of its consequent and assume that each single term in the consequent part of each new rule is expressed by the necessary conjunction $\wedge$ of several low-order terms. Now we give to c_i the meaning of conditions (facts) that must be satisfied and to a_i the meaning of actions that must be performed if the conditions are satisfied. The c_i represent the left-hand side (LHS) of r, a_i the right-hand side (RHS).
- The system also includes a working memory (WM) where we store the facts that are submitted as input to the system or that are inferred by the system itself while it functions.

While it functions, the system repeatedly performs a "recognize-act" cycle, which can be characterized as follows in the case of conventional expert systems (condition-driven ESs, see later):

- In the selection phase, for each rule r of the rule base, the system (1) determines whether LHS(r) is satisfied by the current WM contents, that is, if LHS(r) matches the facts stored in the WM (match subphase), and, if so, (2) adds the rule r to a particular rule subset called the conflict set (CS) (addition subphase). When all the LHS are false, the system halts.
- In the conflict resolution phase, a rule of the CS is selected for execution. If it is impossible to select a rule, the system halts.
- In the act phase, the actions included in RHS(r) are executed by the interpreter. This is often called "firing a rule." Firing a rule normally changes the content of WM and possibly the CS. To avoid cycling, the set of facts (instantiation) that has instantiated the LHS variables

of the fired rule becomes ineligible to provoke the firing of the same rule, which, of course, can fire again if instantiated with different facts.

A schematic representation of the recognize-act cycle is given in Fig. 8. The name conflict set results from the fact that, amongst all the competing selected rules that agree with the current state of WM, it is necessary to choose the only one to be executed by the interpreter in the current cycle. Choosing and executing multiple rules is possible in theory but very impractical in practice. The specific strategy chosen to resolve the conflicts depends on the application and can be relatively complex, because the execution of a rule may lead other rules to "fire" or, on the contrary, it may prevent the execution of other rules. Then it is possible to use user-defined priorities. The user is allowed to choose a particular approach, such as giving preference to rules that operate on the most recent information added to WM or that match the highest number of items, or to the most specific rule, the one with the most detailed LHS that matches the current state of WM. Otherwise, it is possible to use predefined criteria for ordering that may be static (i.e., a priority ordering is assigned to the rules when they are first created) or dynamic.

This type of architecture is at the origin of a very important property of production systems: the independence of *knowledge* from the *control* of how the knowledge is applied. Each set of rules making up a particular knowledge base is created totally independently from the control structure. Each rule in the set must express a relationship between LHS and RHS which must hold a priori in a static way. In other words, the validity, the "truth" of the rule must subsist independently of when it is applied. Comparing with conventional programming techniques, we can also say that, in a production (or, more generally, rule-based system), a change in the knowledge base is not propagated throughout the program as a change in a procedural program can be. This means also that the LHS must express, at least in principle, all of the necessary and sufficient conditions that allow the RHS to be applied.

Production systems can be classified into two different categories according to the way the rules are compared with the data of WM. In the conventional production systems, the comparison is between LHS(r) and WM as illustrated previously (condition-driven, or forward-chaining systems). But is also possible to compare RHS(s) with WM (action-driven, or backward-chaining systems). In this last case that we have taken, Eq. (14) is generally representative of the production rules:

$$c_1 \wedge c_2 \ldots \wedge c_n \rightarrow a_1 \wedge a_2 \ldots \wedge a_m \tag{14}$$

is used in a way that coincides particularly well with the interpretation of logical clauses as implications. The a_i, for example, act as the subgoals to be satisfied to prove the condition. Then we can say that logic programming and PROLOG and DATALOG in particular, work by backward-chaining from a goal.

Generally we can say that the condition-driven, forward chaining production systems are useful in dealing with large rule sets, where the number of possible goal states is very high and it is impossible to select some "best goal" a priori. Then it is better to deal with the data opportunistically, as they arrive in the environment of the system, and to be driven by the data towards a suitable goal. The action-driven, backward-chaining production systems allow implementing more efficient and more focused strategies. In these systems, a goal G is chosen—in its initial state, WM is reduced to G—and the system selects all of the rules that may lead to G, that is, all of the rules where G appears among the a_i of the RHS. If several rules are selected, again we have a CS nonempty and a conflict resolution problem. In the act phase, the c_i in the LHS of the fired rule are chosen as the new subgoals. They are added to WM, and a new recognize-act cycle begins. The process continues until all of the inferred subgoals are satisfied. The efficiency is linked with the fact that the rules are selected in a sequence which proceeds toward the desired goal.

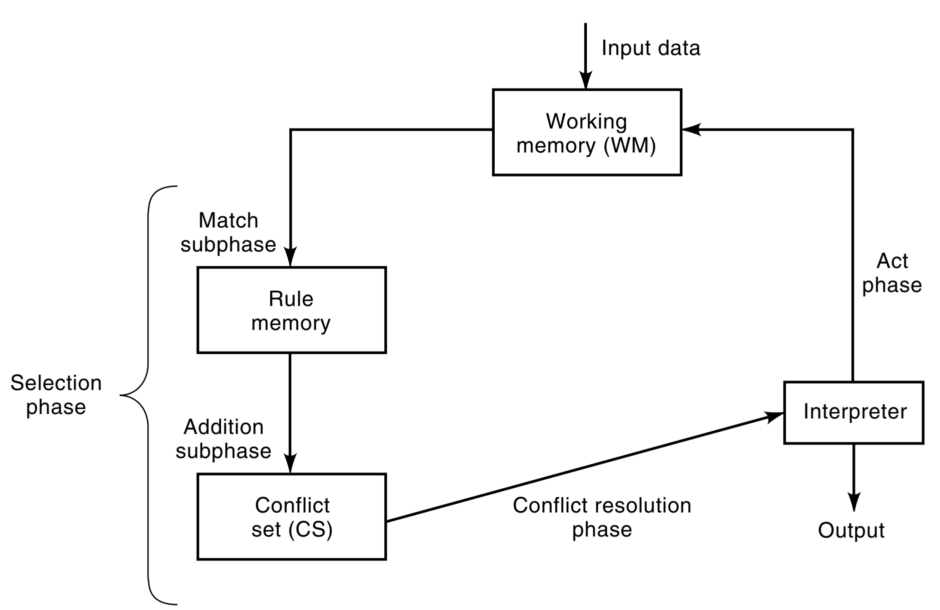

Figure 8. A schematic representation of the recognize-act cycle for an expert system using a set of production rules.

```
Rule 88 IF :
        1) the infection type is primary-bacteremia, and
        2) the site of the culture is one of the sterile sites, and
        3) the suspected portal of entry of the organism is the
           gastro-intestinal tract
     THEN :
        there is suggestive evidence (0.7) that the identity of the
                          organism is bacteroids.
```

Figure 9. An example of MYCIN's rule. MYCIN is a backward-chaining system. The aim of rule 88 is to deduce the simultaneous existence of the facts "the infection type is primary bacteremia", "the suspected entry point is . . .", etc., from the assertion "there is evidence that the organism is bacteroides."

Now to give an example of an actual rule, we propose in Fig. 9 the English version of a production rule, "rule 88", which is part of about 500 rules used in one of the best known and historically important expert systems the MYCIN system; see Ref. 55. MYCIN, built up in the mid 1970s, was designed to perform medical diagnosis (prescribe antibiotic therapy) in the field of bacterial infections, based on medical knowledge of approximately 100 causes of infection buried in its rules. The MYCIN system was a backward-chaining system, that is, the aim of rule 88 was to deduce from the assertion "there is evidence that the organism is bacteroids", the simultaneous existence of the facts "the infection type is primary bacteremia," "the suspected entry point is . . .," etc.

Additional Technical Details. The numeric value that appears in the RHS of rule 88 is a certainty factor (CF), a way of estimating belief in the conclusions of a rule-based system that has been popularized by MYCIN. We can say that the presence of the CFs constitutes the main difference between a simple production system and a real expert system (ES) and, a fortiori, between a logic programming system and an expert system. Through the CFs and other more sophisticated mechanisms (see below), ESs can express, even very roughly, the *uncertainty* linked with a given assertion, instead of, as in PROLOG and DATALOG, affirming that all of the assertions are simply true or false (see the analogous remarks that are at the origin of the fuzzy logic systems). Another important difference of an ES with respect to a simple rule system concerns the possibility for an ES to provide a sort of explication of its behavior. This can be obtained by printing the chain of rules that have led to a given conclusion and by using of the fact that each rule expresses directly the information on which its particular deduction is based and the reasons why this deduction holds.

A CF varies in value between -1 and $+1$. If the value is zero, this means that there is no evidence for the hypothesis being examined. When the value of CF is > 0, and is moving toward $+1$, this means that evidence increasingly supports the hypothesis. When CF < 0, and is moving toward -1, the hypothesis is increasingly unsupported by the evidence. An important point here is that CFs, like the fuzzy sets, are *not* probabilities. They do not deal with the dependence/independence problems typical of probabilities, and moreover, they are defined and combined through a very ad hoc system of rules.

The CFs associated a priori (off-line) with the rules of a production system can be modified when the rules are chained together during functioning of the system. Because the rules fire according to the recognize-act cycle of Fig. 8, there is a sort of propagation of the CFs down the inference chain that results in an increase, decrease or stabilization of the different CFs encountered along the chain. The modifications are executed according to the ad hoc rules suited to the certainty factor theory. Among them, three sorts of rules are particularly important, the parallel combination rule, the propagate changes rule, and the Boolean combination rule.

The first is used when several rules (at least two) are characterized by the presence of *sure* (but distinct) LHSs—that is the LHSs are *facts* that, as in the LHS of the Rule 88 before, are not affected by any sort of uncertainty—and asserting the *same* RHSs which are, however, characterized by *different* CFs according to the different rules. Indicating with u and v the CFs associated, respectively, with the RHS of two rules r_1 and r_2, then $r_1 \equiv \text{LHS}_1 \rightarrow \text{RHS},u$; $r_2 \equiv \text{LHS}_2 \rightarrow \text{RHS},v$. To reuse the (identical) RHS in the chain of deductions, it must be associated with a new CF, w. This last depends on the signs of u and v:

$$u,\ v > 0 \Rightarrow w = u + v - uv$$
$$u < 0 \vee v < 0 \Rightarrow w = \frac{(u + v)}{[1 - \min(|u|,\ |v|)]}$$
$$u,\ v < 0 \Rightarrow w = u + v + uv$$

The propagate changes rule modifies the CF associated with RHS(r) when the rule r itself is uncertain, that is, as the result, for example, of a chain of inferences. In this case, LHS(r) is as well associated with a CF. If the rule r now is LHS(r),u $\rightarrow$ RHS(r),v, the new CF w associated with RHS(r) is

$$w = v \max(0,\ u)$$

Finally, the Boolean combination rule must be used when LHS(r) is, as usual (see rule 88 before), a Boolean combination of literals. The CF w resulting from the "and" and "or" combinations of two LHS literals l_l and l_l, characterized, respectively, by the CFs u and v, are

$$l_1,\ u \wedge l_2,\ v \Rightarrow w = \min(u,\ v)$$
$$l_1,\ u \vee l_2,\ v \Rightarrow w = \max(u,\ v)$$

This means that, if the literals (predicates) are "anded," the lowest value CF is propagated in the LHS. If they are "ored," on the contrary, the maximum value CF is propagated.

The CF approach has several advantages. The main advantages are (1) they are considerably less difficult to evaluate than probabilities and (2) they are independent, so that we can consider their modifications a rule at a time. Independence also means that adding or deleting rules does not imply any remodeling of the entire system of CFs. On the other hand, they can also lead to very strange results, as happens when the number of parallel rules supporting the same hypothesis is high. In this case, for example, the application of the parallel combination rule produces CFs that systematically approach one even in the presence of small values for the original CFs; see (Ref. 56, p. 562). Moreover, the results of applying of the above rules are *monotonic,* that is, the CFs cannot be adjusted if some facts used in the processing are later retracted. The Dempster–Shafer approach, (57,58), has more reliable mathematical foundations than the CF approach, even if it is neatly more complex from a computational point of view. It is based on the idea of adopting a combination calculus where, given a set of hypotheses, all of the possible combinations in the hypothesis set are considered, and includes both the classical Bayesian approach and the CF approach as special cases. The actual tendency seems, however, to ground uncertain reasoning techniques for knowledge-based-systems (KBSs) generally on Bayesian probability theory; see again Ref. 58 and the article BELIEF MAINTENANCE for the so-called Bayesian belief networks, a graphical data structure that exploits conditional dependencies (causal relationships) between events to represent the joint probability distribution of a problem domain—an arc from node A to node B means that the probability value of A has a direct effect on the probability of B.

Now we conclude this section by mentioning the RETE algorithm. Returning to the differentiation between backward chaining and forward chaining, forward chaining often involves dealing with a large quantity of data and a large set of rules. Unlike backward chaining systems where the goal-directed reasoning guides the execution of the rules, in a forward chaining system every fact entered into WM must be compared with every LHS of every rule, leading to a number of combinations that became unmanageable without some mechanism to improve efficiency. The RETE algorithm, developed by Charles L. Forgy in the mid 70s (59) and inserted in the OPS5 production rules language, allows speeding up this heavy matching process. OPS5 is one of the most popular tools for developing ESs according to the production rule paradigm. Its latest version, OPS/R2, supports both forward and backward chaining and objects with inheritance. See Ref. 60 for an historical overview of the development of the OPS language.

To understand more precisely the need for such a mechanism, it is necessary to understand exactly the modalities of constructing the conflict set (CS). Suppose that a fact of WM is used to instantiate a rule r_1 and that the firing of another rule r_2 produces the deleting of this fact. This modifies the conditions under which the LHS of r_1 has been instantiated, and the rule r_1 must be suppressed from the conflict set. That is to say, the conflict set must be recreated for every cycle by examining all of the rules and producing for each of them a list of all the possible instantiations according to the contents of the working memory. This process is particularly inefficient because, in most production systems, WM changes slowly from cycle to cycle (less than 10% of the facts are changed in a cycle). This means that a majority of the production rules are not affected by changes with respect to their instantiations and that a program that reiterates the construction of the conflict set on each cycle probably repeats the same large number of operations, again and again. The main idea of the RETE algorithm is saving the state of the CS at the end of a given cycle and generating in the next cycle only a list of the changes to be incorporated to the CS as a function of the changes that have affected the WM. Then the pattern matcher can be viewed as a *black box* where the input is labelled as Changes to WM and the output Changes to CS (59, p. 22). In practice, the black box is implemented as a dataflow graph (the RETE network). A very clear description of a practical implementation of RETE algorithm is given in Ref. 61.

Representation and Inference by Inheritance

Inheritance is one of the most popular and powerful concepts used in the artificial intelligence domain. At the same time it has very high value *at least* as

- a *static* structuring principle that allows grouping similar notions in classes and economizing in the description of the attributes of the entities at the lower levels of a class, given that they can be inherited from the high-level entities;
- a *dynamic* inferencing principle that allows deductions about the properties of the low-level entities because these properties can be deduced from those that characterize the high-level entities. In this context, some well-known problems are linked with the fact that, for example, penguins and ostriches pertain to the class birds, but they cannot inherit the property "can_fly" from the description of this general class.
- a *generative* principle that allows defining new classes as variants of the existing classes. The new class inherits the general properties and behavior of the parent class, and the system builder must specify only how the new class is different.

In an AI domain, the inheritance principle is normally used to set up hierarchies of concepts, ontologies, to use an up-to-date and very fashionable term, (62,63). The intuitive idea of concept is not easy to define very precisely. As a useful approximation, we can say that we can think of concepts in the context of a practical application as the important notions that it is necessary to represent to obtain correct modeling of the particular domain examined. Moreover, the most general among them, like *human_being* or *physical_object,* are common to a majority of domains. Concepts in AI correspond to classes in object-oriented representations, and to types in the standard procedural programming languages. In this introductory section, we deal mainly with the general architectural issues related to constructing well-formed hierarchies of concepts. In the next two sections we examine the specific issues concerning the internal structure of the concepts, that is, how the attributes (properties, roles, slots etc.) that characterize a given concept are represented.

The main conceptual tool for building up inheritance hierarchies is the well-known IsA link, also called AKindOf (ako), SuperC, etc. (see Fig. 10). We attribute to IsA, at least for the moment, the less controversial and plain interpretation [see (64)] saying that this link stands for the assertion that *concept_b* (or simply *B*) is a specialization, IsA, of the more general *concept_a* (*A*). This sort of relationship is normally expressed in logical form as

$$\forall x(B(x) \supset A(x)) \tag{15}$$

This expression means that, if any *elephant_* (*B*) IsA *mammal_* (*A*) and if clyde_ is an *elephant_,* then clyde_ is also

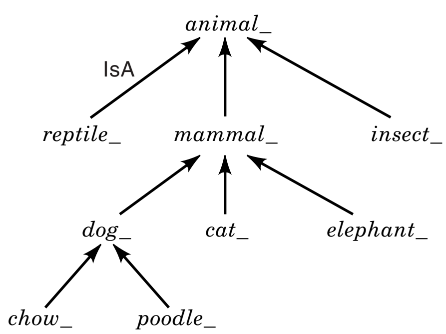

Figure 10. An example of a simple inheritance network where the concepts are linked by IsA links.

a *mammal_* (Here we adopt the convention of writing the *concepts_* in italics, and their instances_ (individuals) in roman characters). When this law is interpreted in a strict way, it also implies that the instances of a given concept *B* must inherit *all* the features (properties) of *all* the more general concepts in the hierarchy that have *B* as a specialization. This law is called the "strict inheritance" law (it often admits exceptions; see later). An important aspect of the semantic interpretation of the inheritance hierarchies consists of the fact that the ordering relation giving rise to the algebraic structure of the inheritance hierarchies is the "property inclusion" or entailment. The property exclusion also intervenes in the definition of the semantics of the well-formed hierarchies by assuming that the siblings immediately descended from the same parent node are mutually exclusive. This means that, if we consider that *dog_, cat_* and *elephant_* are all siblings deriving directly from *mammal_,* the properties (obviously, not inherited from *mammal_*) that characterize them as separate concepts must all be mutually exclusive.

The clyde_ example allows us to introduce the indispensable complement of IsA to construct well-formed inheritance hierarchies, the InstanceOf link. Note that the awareness of this necessity is a relatively recent acquisition of the knowledge representational domain. In the eighties, several operating systems (commercial and not) based on inheritance mechanisms still could not distinguish between concepts and instances of the concepts. A well-known example in this context is that of KEE, see Ref. 65, one of the early and most powerful commercial environments for developing complex knowledge-based systems (KBSs); see also the following sections. Followers of a uniform approach in which all the units, to adopt the KEE terminology have the same status claim that, for many applications, this distinction is not very useful and only adds all sorts of unnecessary complications (see later).

The difference between *B* IsA *A* and C InstanceOf *B* is normally explained in terms of the difference between the fact that *B* is a subclass of *A* in the first case, operator $\subset$, and that C is a member of the class *B* in the second, operator $\in$. Unfortunately, this is not sufficient to eliminate any ambiguity about the notion of instance, and this last notion is, eventually, much more controversial than the notion of concept. The main problems involve: (1) the possibility (or not) of accepting all of the concepts of an inheritance hierarchy to the exclusion of the root as instances of higher level concepts, instead of deciding that the instances can only be derived nodes strictly independent of the concept nodes; (2) even limiting the notion of instance to this last interpretation, there is still an ambiguity about the possibility (or not) of having several levels of instances, that is, instance of instances.

If a very liberal interpretation of the notion of instance is admitted, clyde_ is an instance of *elephant_* but *elephant_* can also be considered, to a certain extent, an instance of *mammal_*. This is accepted, in some object-oriented systems. In this case, the logical properties of the instances are likely to become strongly dependent on the particular choice of primary concepts selected to set up a given inheritance hierarchy. For example, in front of a figure like Fig. 10, we could infer that the InstanceOf relationship is, like IsA, always transitive: if fido_ InstanceOf *poodle_,* it is also, evidently, an instance of *animal_*. But if, in this same figure, we substitute the root *animal_* with the root *species_,* we can still consider that *poodle_* InstanceOf *species_,* but it becomes very difficult to assert fido_ InstanceOf *species_;* see also (Ref. 66 pp. 332–339). Then we prefer adding to the set-oriented definition of an instance a sort of extensional definition in the Woods style (67). We propose considering that all the nodes of a well-formed inheritance hierarchy like that of Fig. 10 must be considered only as concepts, that is, general descriptions/definitions of generic intensional notions, like that of poodle. When necessary, an InstanceOf link can be added to each of these nodes; this link has the meaning of a specific existence predicate. In this way, we will declare that a specific, extensional incarnation of the concept *poodle_* is represented by the individual fido_. Now the introduction of instances becomes a strictly local operation to be executed explicitly, when needed, for each node (concept) of the hierarchy (see also the overriding phenomena later). Another consequence is represented by the fact that, in this way, concepts participate in the inheritance hierarchy directly. Instances participate indirectly in the hierarchy through their parent concepts.

Localizing the introduction of instances considerably clarifies the meaning and the practical modalities of using this notion, but it is not yet sufficient to eliminate any ambiguity. It remains to be decided if the instances are to be systematically considered as terminal symbols, or whether it can be admitted that an instance can be characterized in turn by the presence of more specific instances. The classical example, see Ref. 35, is given by paris_, an individual that is an instance of the concept *city_,* but which could be further specialized by adding proper instances (i.e., viewpoints) like Paris of the tourists, Paris as a railway node, Paris in the *Belle Epoque,* etc. If, for clarity, instances are always considered terminal symbols, viewpoints can be realized according to a solution which goes back to the seminal paper by Minsky about frames (68). This consists of introducing specialized concepts in the inheritance hierarchy like *tourist_city, railway_node, historical_city* that admit the individual paris_ as an instance. Then paris_ inherits from each of them particular, bundled sets of attributes (slots) like {UndergroundStations, TaxisBaseFare, EconomyHotels. . .} from *tourist_city,* {TypesOfMerchandise, DailyCommutersRate. . .} from *railway_node,* etc.

The precise definition of the meaning of InstanceOf is not the only problem that affects the construction and use of inheritance hierarchy, especially when the inheritance considered is more behavioral than structural, that is, more interested in the actual behavior and meaning of the properties inherited than in the pure mechanical aspects of the propagation. From a behavioral point of view, the two main problems are "overriding" (or defeasible inheritance or inheritance with exceptions) and "multiple inheritance."

Overriding consists of admitting exceptions to the strict inheritance law introduced previously. In a strict inheritance world, from fido_ InstanceOf *poodle_*, we could automatically deduce fido_ InstanceOf *mammal_* and fido_ InstanceOf *animal_* (see Fig. 10), without being obliged to assert explicitly when needed, that fido_ is also an instance of *mammal_* and *animal_*. Now consider this group of assertions:

a. Elephants are gray, except for royal elephants.
b. Royal elephants are white.
c. All royal elephants are elephants.

Assertion (c) introduces a new concept, *royal_elephant,* as a specialization of *elephant_* of Fig. 10. Now if clyde_ InstanceOf *royal_elephant,* the strict inheritance law would lead us to conclude that the property (slot) ColorOf of clyde_ is filled with the value *gray_*, but from (a) and (b) we know that the correct filler is instead *white_*. This means that *royal_elephant* has an overriding property, ColorOf or, in other terms, that the property ColorOf of *elephant_* must not be considered as a systematically inheritable property. Then a differentiation at least implicit between overriding properties and nonoverriding properties is introduced in the set of properties (attributes, slots etc.) that characterize a given concept: for *elephant_* and all of its instances and specific terms we can say that FormOfTheTrunk is a nonoverriding property because its associated value is always *cylinder_*. ColorOf is, on the contrary, overriding. We can visualize in Fig. 11 the situation described in the three previous assertions. The crossed line (cancel link) indicates that the value associated with the overriding property ColorOf has been actually changed passing from *elephant_* to *royal_elephant.* Note that, in most of the implemented knowledge-based systems (KBSs), the cancel link is not explicitly implemented, and the overriding can be systematically executed.

In a well-known paper, R.J. Brachman (69) warns about the logical inconveniences linked with introducing an unlimited possibility of overriding. Under the overriding hypothesis, the values associated with the different properties of the

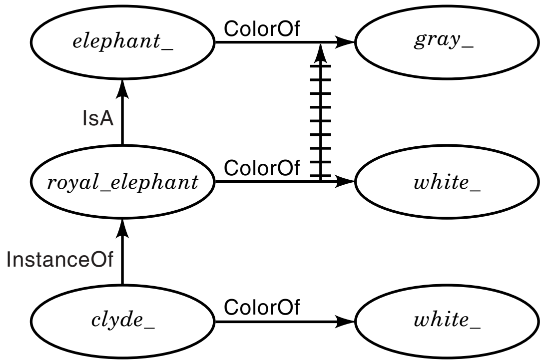

Figure 11. Overriding properties: in this figure, the crossed line (cancel link) indicates that the value associated with the overriding property ColorOf has been changed passing from *elephant_* to *royal_elephant.*

concepts must now be interpreted simply as "defaults" that are always possible to modify. Brachman also evokes the possibility that not only the values of the properties, but the properties themselves can be overriden, possibly leaving the values unchanged: a *giraffe_* is an *elephant_* where the value *cylinder_* associated with the property TrunkOf of *elephant_* does not change, but the property itself has been overridden, and it is now called NeckOf for *giraffe_* (69, pp. 85–86). As a consequence, now it becomes impossible to use the internal structure of the different concepts, that is, the presence of particular properties and values, to determine if a given concept is more general or more specific than another and then to determine automatically the position of a new concept in the inheritance network. Moreover, given that all the properties of the concepts are now purely local, any concept acts as a *primitive* whose properties must be explicitly asserted each time (69). Then the benefits associated with using the inheritance hierarchies seem close to vanishing. Without completely endorsing such catastrophic conclusions, it appears clearly that an uncontrolled amount of overriding can introduce some really serious coherence problems.

Giving, however, that dealing with exceptions is an evident necessity in the knowledge representational domain, AI researchers have tried to avoid the danger of uncontrolled use of overriding techniques by using some form of nonclassical logic to provide formal semantics for inheritance hierarchies with defaults. For example, both Etherington (70,71) and Nado and Fikes (72) use Reiter's Default Logic with this aim (73,74). Very briefly, a *default theory* is a pair (D, W) where D is a set of "default rules" (seen as a sort of inference rules) normally concerning the properties of the concepts, like "Typically, elephants have four legs," and W is a set of "hard facts" like "All elephants are mammals" (or "Margaret Mitchell wrote 'Gone with the Wind'"). Formally, W is a set of first-order formulas, whereas a typical default rule of D can be denoted as

$$\frac{\alpha(x_1, \ldots, x_n) : \beta(x_1, \ldots, x_n)}{\gamma(x_1, \ldots, x_n)} \tag{16}$$

where α, β, and γ again are first-order formulas whose free variables are among $x_1, \ldots, x_n$. The notation $\omega(x_i)$ is an abridged logical-like notation to express generally that x_i IsA ω, x_i InstanceOf ω, ω PropertyOf x_i. Informally then, a rule like Eq. (16) means that for any individuals $x_1, \ldots, x_n$, if $\alpha(x_1, \ldots, x_n)$ is inferable, and if $\beta(x_1, \ldots, x_n)$ can be consistently assumed, then infer $\gamma(x_1, \ldots, x_n)$. For our previous example concerning royal elephants, Eq. (16) becomes

$$\frac{elephant\_(x) : gray\_(x) \wedge \neg royal\_elephant(x)}{gray\_(x)}$$

From the previous definition, it can be seen that, if we assume simply clyde_ InstanceOf *elephant_*, we can say that clyde_ ColorOf *gray_* and ¬ clyde_ InstanceOf *royal_elephant* are consistent with this assumption. Hence, clyde_ ColourOf *gray_* can be inferred. In logical notation, from the initial assumption *elephant_*(clyde_) and having verified that *gray_*(clyde_) and ¬ *royal_elephant*(clyde_) are consistent with the assumption, we can infer *gray_*(clyde_). On the other hand, if the initial assumption now is *royal_elephant*(clyde_), using the hard fact *royal_elephant* IsA *elephant_* (see Fig. 11), we are reduced again to the situation of the previous example, that is *elephant_*(clyde_). In this case, however, the consistency condition $\beta(x_1, \ldots, x_n) = gray\_(x) \wedge \neg royal\_elephant(x)$ is violated given the initial assumption *royal_elephant*(clyde_) that blocks the default rule, then preventing the derivation of *gray_*(clyde_).

The inheritance hierarchy of Fig. 10 is a tree. Each node (concept) has only one node immediately above it (its parent node) from which it can inherit the properties. In this case, the mode of transmission of the properties is called single inheritance. Normally, however, in real-world inheritance hierarchies, a concept can have multiple parents and can inherit properties along multiple paths. For example, *dog_* of Fig. 10 can also be seen as a *pet_,* then inheriting all of the properties of the ancestors of *pet_,* pertaining maybe to a branch *private_property* of the global inheritance hierarchy. This phenomenon is called multiple inheritance. Now the inheritance hierarchy becomes a "tangled hierarchy" as opposed to a tree, a partially ordered set (poset) from a mathematical point of view. We note here that the inheritance hierarchies admitting multiple inheritance can be assimilated with the standard form of semantic networks (33).

Multiple inheritance contributes strongly to the simplification of the inheritance hierarchies by eliminating the need for duplicating some concepts and the corresponding instances that would be necessary to execute to reduce the hierarchy to a simple tree. A possible example of duplicated concepts could be *dog_as_valuable_object* and *dog_as_carnivore_mammal.* The use of the multiple inheritance approach, however, can give rise to conflicts about the inheritance of the values associated with particular properties.

To illustrate this problem, we use an example that relates to one of the most intricate issues in constructing well-formed ontologies, the classification of "substances" (35,66). According to a majority of researchers, concepts like *substance_* and *color_* are to be regarded as examples of "nonsortal concepts." The sortal concepts correspond to notions that can be directly materialized into enumerable specimens (i.e., instances), like chair or lump (which correspond to physical objects). Nonsortal concepts cannot be directly materialized into instances. Note that a notion like white gold is a specialization of gold, not an instance. Now let us consider Fig. 12, that can be viewed as a first, rough solution to the problem of correctly classifying a notion like nuggets of gold. The entire situation of course, is highly schematized. This notion corresponds certainly to *physical_object*—and, because of this fact, it admits the existence of direct instances, gold_nugget_1, gold_nugget_*n*, etc. On the other hand, it can also be considered, to a certain extent, a specialization of *gold_* because it inherits at least some intrinsic properties, like ColorOf, MeltingPoint, etc., and the corresponding values. In adopting a solution like that of Fig. 12, however, an explicit inheritance conflict appears, because according to the organization adopted in this figure, *gold_nugget* may inherit both the values "yes" and "no" for the property HasInstances.

A multiple inheritance conflict can be resolved by two basic techniques. In the first, implicit technique, a precedence list is computed mechanically by starting with the first leftmost concept that represents a generalization (superconcept) of the concept where the conflict has been observed. In the case of Fig. 12, the leftmost immediate superconcept of *gold_nugget*

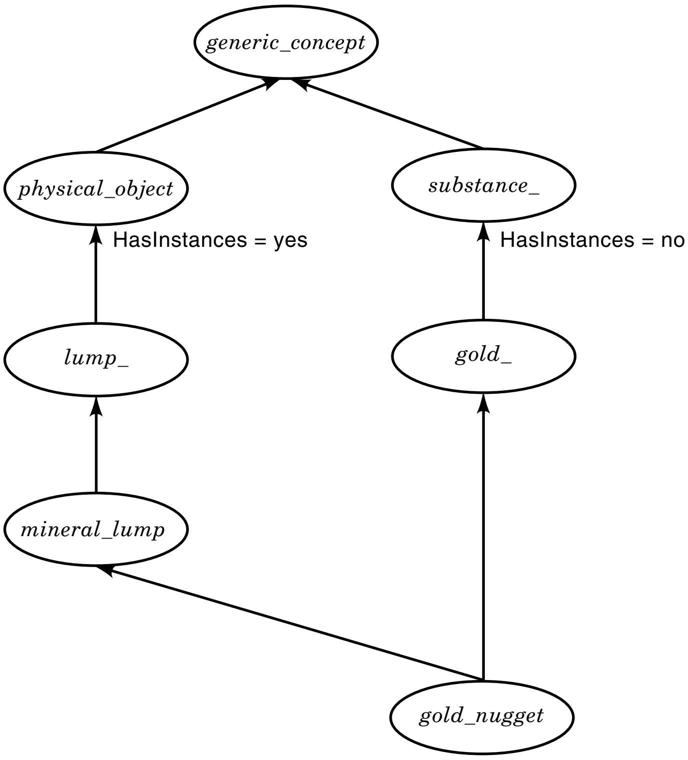

Figure 12. Inheritance conflict because *gold_nugget* may inherit from both *physical_object* and *gold_* (a *substance_*).

is *mineral_lump.* Constructing the precedence list proceeds by visiting depth-first the nodes in the left branch, then those of the right branch, then the join, and up from there. In our example, this list is (*mineral_lump, lump_, physical_object, gold_, substance_, generic_concept*). Then *gold_nugget* inherits the properties of *mineral_lump,* including HasInstances = yes, as required.

Obviously, this technique strictly depends on the particular arrangement adopted in the constructing the inheritance hierarchy and can oblige one to insert a number of dummy concepts (analogous to the "mixins" of object-oriented programming) to establish a correct precedence list. The second technique for dealing with conflict resolution is an explicit technique that attributes to the user the responsibility of specifying from which superconcept a given conflicting property must be inherited. Advanced environments (knowledge engineering software environments (KESEs), see the next sections) for the setup of large KBSs, like Knowledge Craft or ROCK by the Carnegie Group Inc., allow for a particularly neat implementation of this principle. They supply the user with tools for specifying exactly the inheritance semantics for the properties of a given concept, that is, the information passing characteristics that indicate which slots and values must be included, excluded, introduced further, or transformed during inheritance ('mapped'). In this way, when defining the properties of *gold_nugget,* it becomes very easy to require that this last concept is a specialization of *mineral_lump* (and, therefore, of *physical_object*) and that it only inherits the set of intrinsic properties from *gold_.* The new arrangement is depicted in Fig. 13, which gives a more precise representation of the relationships between the concepts involved in this (very stereotyped) situation.

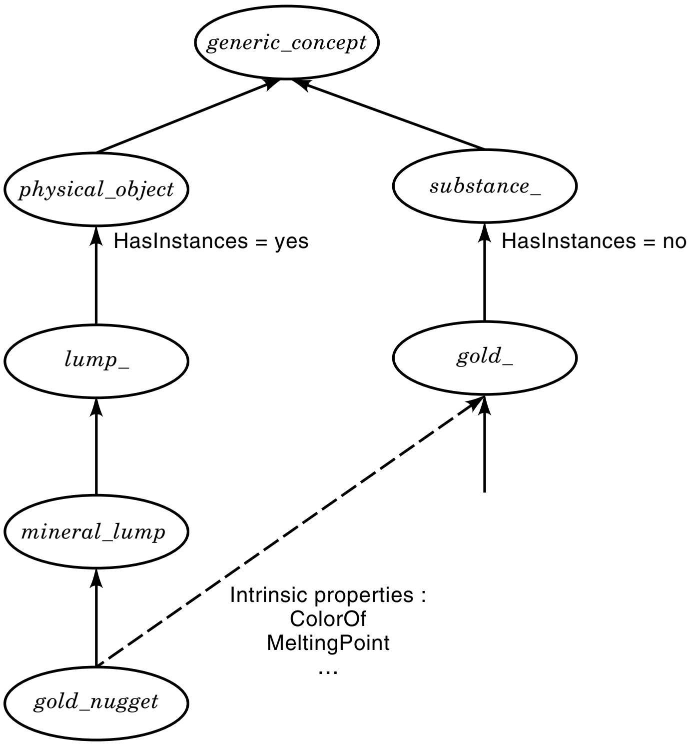

Figure 13. Resolution of the inheritance conflict requiring that *gold_nugget* (a specialization of *mineral_lump* and, therefore, of *physical_object*) inherits only the set of intrinsic properties from *gold_*.

When defeasible inheritance (materialized by the presence of cancel links) and multiple inheritance combine, we are confronted with very tricky situations like the notorious Nixon Diamond (Fig. 14). In this version of the Diamond, the most frequently used, we admit that it is possible to have an individual, nixon_, as a common instance of two different concepts, *republican_* and *quaker_*. Several inheritance-based systems do not allow this possibility. Postulating, however,

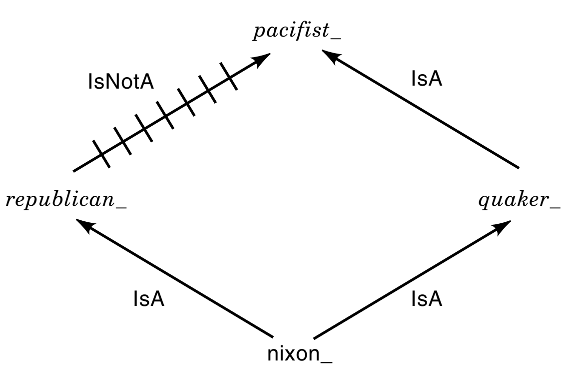

Figure 14. The Nixon Diamond. Asking: "Is Nixon a pacifist or not?", we are in trouble given that, as a Quaker, Nixon is (typically) a pacifist but, as a Republican, Nixon is (typically) not a pacifist. With a skeptical attitude, we refuse to draw any opinion as to whether or not Nixon is a pacifist. With a credulous attitude, we will try to deduce as much as possible, then generating all of the possible extensions of the ambiguous situation, and then we will generate both the solutions, *pacifist_* and ¬ *pacifist_*.

the presence of an intermediate concept like *republican_having_quaker_convictions,* which specializes both *republican_* and *quaker_* and to which we could attach the nixon_ instance, would not change the essence of the problem. If we ask now: Is Nixon a pacifist or not?, we are in trouble given that, as a Quaker, Nixon is (typically) a pacifist but, as a Republican, Nixon is (typically) not a pacifist. Then a reasoner dealing with this situation must choose between two possible attitudes. A reasoner with skeptical attitude will refuse to draw conclusions in ambiguous situations and, therefore, will not opine whether or not Nixon is a pacifist. With a credulous attitude, the reasoner will try to deduce as much as possible, generating all of the possible extensions of the ambiguous situation. In the Nixon Diamond case, both solutions, *pacifist_* and ¬ *pacifist_*, are generated.

This problem (and the similar ones) has given rise to a flood of theoretical work without leading to real, definite solutions. Among the classics, we mention Refs. 75–77. Padgham (78) proposes some solutions to the Nixon Diamond quandary by introducing specific assumptions about the concepts to be considered, for example, by stating that only *typical* Quakers are pacifist or, on the contrary, that Quakers are *always* pacifist. This approach confirms a well-known principle that to obtain sound solutions to the most complex knowledge representational problems, the possibility of disposing of large amounts of domain knowledge is at least as important as the conception of clever formal solutions.

FRAME SYSTEMS AND KNOWLEDGE ENGINEERING SOFTWARE ENVIRONMENTS

In the previous section, we considered concepts characterized solely by (1) a conceptual label (a symbolic name); (2) hierarchical relationships with other concepts that are symbolized by the IsA links. In reality, the main reason for the persistent popularity of frame-based systems in industry and research is that these systems allow associate with any concept a "structure" often very naturally reflecting the ordered knowledge human beings have about the intrinsic properties of these concepts and the network of relationships, other than hierarchical, which the concepts have with each other. Then the current interpretation of frames corresponds to a sort of "assertional interpretation" that views frames as knowledge representational systems able to represent a collection of statements about the important notions of a given application domain. This interpretation only partly coincides with the original motivations behind the introduction of frames by Minsky (68). These can be roughly summarized by saying that, for him, the utility of frames consisted mainly of the possibility of using them to semantically direct the reasoning of scene-analysis systems by instantiating the descriptions of stereotypical situations. For the well-known debate initiated by Hayes about the possibility of fully reducing frames to a simple notational variant of first-order logic (FOL) see Refs. 79 and 80.

A frame is basically a named data structure, very similar to the "objects" of the database domain (see DATABASE LANGUAGES) that includes a flexible collection of named "slots" (or fields, roles, etc.) which can be associated with "values." Usually, the slots are distinguished from their fillers (values); see, however, Ref. 81. Usually there is no fixed number of slots nor a particular order imposed on these slots. The slots may be accessed by their names; they generally represent the important "properties" necessary to introduce to characterize a given concept completely. Unfortunately, a definition like this is too vague and imprecise to constitute a valid direction for creating the "correct" set of slots. The arbitrariness linked with the subjective choice of the slots in the frame systems has been criticized often (82). In some powerful knowledge representational tools conceived to facilitate constructing complex frame-based systems, like Knowledge Craft or ROCK already mentioned, this arbitrariness is (very partially) obviated by using metastructures that describe precisely the computational behavior of a given slot. For example, in the previous tools, a "slot-control schema", a particular structured object containing information about the properties of a specific slot, for example, the restrictions concerning the domain, the range and the cardinality, the inheritance specifications, etc., can be added to each slot as a sort of formal definition. In other frame-oriented languages, like Knowledge Engineering Environment (KEE) by Intellicorp, this function is assigned to the "facets" that represent annotations on the slots. Like the slots, facets can have values. Facets are normally used to specify a slot constraint, a method for computing the value of a slot, or simply to introduce some documentation string about the slot itself (DOCUMENTATION facet). The most commonly used facets are those used to specify a type restriction on the values of the slot (VALUE-TYPE facet) and to specify the exact number of possible values that a slot may take on (CARDINALITY facet). Some facets on the **PREMISE** and Procedure 1 slots are shown later in Fig. 18.

A relatively clear understanding of the functioning of a set of slots, however, can be obtained by using a sort of "functional" definition. For example, in the (standard) frame component of Narrative Knowledge Representation Language (NKRL), see Ref. 35 for a more complete description, the slots are grouped in three different classes, relations, attributes, and procedures. A general schema of a frame representing an NKRL concept or individual is represented in Fig. 15. Object IDentifier (OID) stands for the symbolic name of the particular concept or individual.

The slots of the relation type are used to represent the relationships of an NKRL frame, concept or individual, to

```
{ OID
     [ Relation  (IsA   |   InstanceOf  :
                  HasSpecialization   |   HasInstance :
                  MemberOf   |  HasMember  :
                  PartOf   |  HasPart  :)
                 (UserDefined1 :
                  ...
                  UserDefinedn : )
       Attribute (Attribute1  :
                  ...
                  Attributen  : )
       Procedure (Procedure1  :
                  ...
                  Proceduren  : ) ] }
```

Figure 15. A general schema of frame where the slots are grouped in three different classes, relations, attributes, and procedures, according to a sort of functional organization. Object IDentifier (OID) is the symbolic name of the particular concept or individual defined by the frame.

other frames. These slots represent the privileged tools to set up complex systems of frames. NKRL provides for eight general system-defined relationships: IsA, and the inverse HasSpecialisation; InstanceOf, and the inverse HasInstance; MemberOf (HasMember) and PartOf (HasPart). IsA and InstanceOf have been discussed at length in the previous section. MemberOf and PartOf correspond, respectively, to the Aggregation and Grouping relationships that, with Generalization (IsA), characterize the semantic models in the database domain. Some of the properties of the direct relationships are shown in Fig. 16.

Note that because of the definitions of concept and instance given in the previous section and of the properties of IsA, InstanceOf, PartOf and MemberOf illustrated in Fig. 16, a concept or an individual (instance) cannot use the totality of the eight relations. More exactly,

- The relation IsA, and the inverse HasSpecialisation, are reserved to concepts.
- HasInstance can be associated only with a concept, and InstanceOf with an individual (i.e., the concepts and their instances, the individuals, are linked by the InstanceOf and HasInstance relations).
- Moreover, MemberOf (HasMember) and PartOf (HasPart) can be used only to link concepts with concepts or instances with instances, but not concepts with instances.

Note also that, in NKRL as in many other semantic network systems (in the widest meaning of these words) (83), only two meronymic relations, MemberOf and PartOf (and of their inverses), are included among the system-defined relationships. The basic criterion for differentiating between the two is the homogeneity (HasMember) or not (HasPart) of the component parts. Moreover, PartOf is characterized by a sort of functional quality—see, for example, "a handle is part of a cup"—that is absent in MemberOf. As is well known, six different meronymic relations are defined in Ref. 84. component/integral object (corresponding to PartOf in NKRL), member/collection (corresponding to MemberOf); portion/mass; stuff/object; feature/activity; place/area. Note that Ref. 84 is still the best reference paper for people interested mainly in the practical implications of using of meronymic concepts. For an overview of some more theoretical (and description logic-oriented) approaches, see Ref. 85. The justification of the NKRL (and similar systems) approach is twofold:

- A first point concerns the wish to keep the knowledge representational language as simple as possible. In this context, the only "relations" which are absolutely necessary to introduce (in addition, of course, to IsA, InstanceOf, and their inverses) are MemberOf and HasMember. In NKRL, for example, they are systematically used to represent plural situations (35).
- On the other hand, dealing systematically with the examples of non-NKRL relations given by Winston and his colleagues in Ref. 84 by using only the existing NKRL tools leads to results which are not totally absurd, even if, sometimes, some aspects of the original meaning are lost. For example, "this hunk is part of my clay" (portion/mass) can also be interpreted as a MemberOf relation,

$(A \text{ IsA } B) \wedge (B \text{ IsA } A) \leftrightarrow A \equiv B$
$(A \text{ IsA } B) \wedge (B \text{ IsA } C) \rightarrow (A \text{ IsA } C)$ {IsA is a partial order relationship}
$(A \text{ IsA } B) \wedge (A \text{ IsA } C) \rightarrow \exists D\ (B \text{ IsA } D) \wedge (C \text{ IsA } D)$
$(A \text{ PartOf } B) \rightarrow \neg\ (B \text{ PartOf } A)$
$(A \text{ PartOf } B) \wedge (B \text{ PartOf } C) \rightarrow (A \text{ PartOf } C)$
$(A \text{ IsA } B) \wedge (B \text{ PartOf } C) \rightarrow (A \text{ PartOf } C)$
$(A \text{ IsA } B) \wedge (A \text{ PartOf } C) \rightarrow (B \text{ PartOf } C)$
$(B \text{ IsA } C) \wedge (A \text{ IsA } C) \rightarrow (A \text{ PartOf } B)$
$(A \text{ IsA } B) \wedge (B \text{ MemberOf } C) \rightarrow (A \text{ MemberOf } C)$
$(\text{C InstanceOf } A) \wedge (A \text{ IsA } B) \rightarrow (\text{C InstanceOf } B)$
$(\text{C PartOf D}) \wedge (\text{C InstanceOf } A) \wedge (\text{D InstanceOf } B) \rightarrow (A \text{ PartOf } B)$
$(A \text{ PartOf } B) \wedge (\text{D InstanceOf } B) \rightarrow \exists\text{C}\ (\text{C InstanceOf } A) \wedge (\text{C PartOf D})$

Figure 16. Some of the axioms that define the properties of IsA, InstanceOf, MemberOf and PartOf. MemberOf and PartOf correspond, respectively, to the Aggregation and Grouping relationships that, with Generalization (IsA), characterize the semantic models in the database domain.

like "this tree is part of the forest," given that we can interpret an individual like generic_portion_of_clay_1 as formed by several hunks, hunk_1 . . . hunk *_n* which, like the trees in the forest, are all homogeneous and play no particular functional role (as in the PartOf examples) with respect to the whole, that is, generic_portion_of_clay_1. "A martini is partly alcohol" (stuff/object) can easily be rendered by using the attribute slots (see later). "An oasis is a part of a desert" (place/area) can be regarded as PartOf.

We conclude the discussion of the relation slots by saying that NKRL allows using of specific user-defined relations to enhance the system-defined relations. See in Ref. 35 the use of a user-defined GetIntrinsicProperties to solve the problems of intrinsic properties inheritance discussed in the previous section. In these cases, of course, the properties of the new relation (see Fig. 16) and the inheritance semantics (i.e., the information passing characteristics indicating which slots and values can be inherited over that relation) must be explicitly specified.

The slots of the attribute type are used to represent the characteristic properties of an object. For example, for a concept like *tax_*, possible attributes are TypeOfFiscalSystem, CategoryOfTax, Territoriality, TypeOfTaxPayer, TaxationModalities, etc. The arbitrariness in the choice of the properties to be selected for a given frame is particularly evident for this class of slots. Fillers of the attribute slots can be both of the following

1. Real fillers (instances): this is the normal (but not mandatory) situation for the slot fillers of individuals. For example, the slot filler of the ColorOf slot of the individual rose_27 is velvety_crimson.
2. Potential slot fillers represented by concepts that define the set of legal, real fillers: this is the normal (but not mandatory; see later) situation for the slot fillers of the concepts. For example, the slot filler of the ColorOf slot in the concept *rose_* could be the concept *color_* (a superconcept of *red_*) indicating that, in the individuals which represent the instances of *rose_*, this particular slot can be filled only by instances of *color_*, for example, instances of *red_*, like velvety_crimson. Note that generally the restrictions about the sets of legal fillers can also be expressed by particular combinations of concepts for example, in the KEE formalism, an expression like "(INTERSECTION *human_being* (UNION *doctor_lawyer_*) (NOT.ONE.OF fred_))" designates a class of fillers that are men, can be doctors or lawyers, but cannot be Fred (65, pp. 90, 91).

As an example of the possibility of having slot fillers for the concepts that are not necessarily concepts, we reproduce in Fig. 17 a fragment of Fig. 10 above where the concepts are now associated with their (highly schematized) defining frames. Note, however, that the two fillers male_/female_ could have been replaced by a hypothetical, subsuming concept *sex_*. Among other things, now this figure makes explicit what inheritance of the properties means. Supposing that the frame for *mammal_* is already defined and supposing now we tell the system that the concept *dog_* is characterized by the two specific properties Progeny and SoundEmission, what the frame *dog_* really includes is represented in the lower part of Fig. 17.

The convenience of being equipped with slots of the procedural type is linked with the remark that because frame-based systems are very popular tools for setting up commercial KBSs, they tend to be configured as independent but sufficient knowledge engineering software environments (KESEs) for applications development. Then they must normally provide alternative inferential and representational schemes in addition to the inheritance-based methods and representations. This is the function assigned to the slots of the procedural type which generally provide various ways of attaching to frames procedural information normally expressed by using ordinary programming languages like LISP or C. In the KEE environment, two standard forms of procedural attachment are used, methods and active values (65, pp. 90–92) both derived from research in the object-oriented and database fields. In KEE, methods are LISP procedures, stored in slots of the procedural type identified as message responders, that can respond to messages sent to the frame. The messages must specify the target message-responder slot and must include any argument needed to activate the method stored at that slot. Active values are implemented in KEE under the form of production rules stored in the procedure slots. The rules are invoked when the slot's values are accessed or stored. Then active values in KEE behave like "daemons." The procedure slots and their values implemented under the form of methods or active values can be defined, of course, at the concept level and then inherited by their associated instances (individuals).

Then this way of using the procedure slots to transform the (relatively static) frame systems into real KESEs can be generalized by specializing some of these slots so that they can represent, for example, the CONDITION, CONCLU-

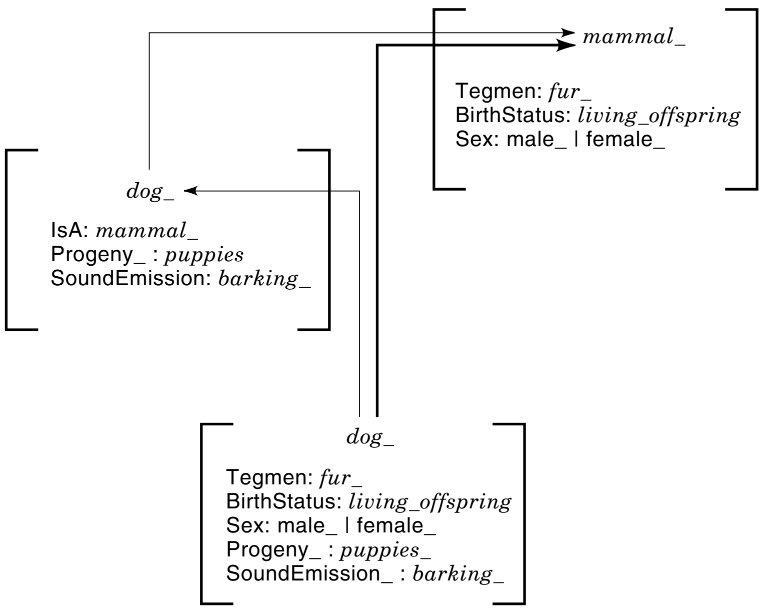

Figure 17. A fragment of the inheritance hierarchy concerning *mammal_* where now the concepts are associated with their (highly schematized) defining frames. The meaning of the locution inheritance of the properties appears here clearly. Supposing that the frame for *mammal_* is already defined, and now supposing we tell the system that the concept *dog_* is characterized by the two specific properties Progeny and SoundEmission, what the frame *dog_* really includes is represented in the lower part of the figure.

SION, and ACTION parts of a production rule (see previous). In this way, production rule systems can be implemented as frame systems with the following advantages:

- Recalling that frames ≡ concepts ≡ classes or types, the fact that each single production rule is realized like a frame means that all of these rules can be easily grouped into classes and that then it is easy to realize powerful indexing schemata to superpose on to the simple, sequential list of rules.
- The reasoning mechanisms suited to the frame systems (mainly, inheritance) may be used to obtain the values needed to instantiate the different parts of the rules, facilitating the task of the production rule inferential engines.

As an example, in Fig. 18 we give two (fragmentary) examples to show the use of frame-like structures in implementing rule-like structures. The first part of the figure displays an example adapted from (Ref. 65, p. 913), which involves using a frame (unit in KEE jargon) BIG.NON.RED.TRUCKS.RULE, specific term (Member) of the class TRUCK.CLASSIFICATION.RULES, to implement a rule for identifying big nonred trucks. Only the slot PREMISE is shown. Note that the wff (well-formed formulas) that constitute this premise are automatically parsed by the KEE system starting from a more readable formulation of this premise expressed by the system builder in a first-order logic language. The second part of the figure is an example adapted from Ref. 86 showing the NKRL representation of the beginning (the topic part of a complex rule) of a normative text (the rule) which corresponds to article n° 57 of the French “General Taxation Law.” See Ref. 35 for some details about the descriptive formalism suited to NKRL. The introduction says, in a rough English translation, “In order to determine the income tax payable by companies which are under the authority of, or which exercise a control over, companies domiciled abroad . . ..” As it appears from Fig. 18, art._57 (the global NKRL representation of the normative rule) is interpreted as an individual instance of the general concept *norms_for_indirect_transfer_of_revenues_abroad*. In this case also the translation from natural language into formal language can also be executed (at least partly) automatically (86,87).

Terminological Logics and the Fundamental Tradeoff

Terminological languages (also called concept or description logic languages), such as KRYPTON (88), NIKL (89), LOOM (90), CLASSIC (91), KRIS (92), and BACK (93), originate in Brachman’s KL-ONE (94), a highly influential knowledge representational system founded on formalization and generalization of the basic principles of frames and semantic networks and intended to permit constructing complex and structured conceptual descriptions. In KL-ONE, the primitives used to represent the internal structure of a concept are called roles which, like the slots in the frame systems, represent the attributes associated with the concepts. Roles (1) supply information about the function of the attribute, that is the intension of the attribute and, moreover, (2) carry the description of the potential fillers, that is the extension (the instances) of the attribute. See also the previous section. The mutual relationships between the roles are managed by the structural description, that is a set of relationships between the role fillers that must hold when the concept and hence the roles are instantiated. The complexity in managing the structural description is one of the most common criticisms of KL-ONE. Concepts are inserted in an inheritance hierarchy (inheritance lattice). To avoid at least some of the “overriding” problems, (see previous discussion), a role is considered a necessary attribute of a concept and, therefore, it is not cancellable. One of the most important contribution of KL-ONE to the theory of knowledge representation is the precise definition of the notion of subsumption. By subsumption, Brachman means that, given a concept D and a SuperConcept C (higher than D in the hierarchy) that subsumes D (i.e., D is a specialization of C), an instance of D will always be, by definition, also an instance of C. In a more concise way, C subsumes D if the extension of D is a subset of the extension of C. Then

```
Unit:  BIG.NON.RED.TRUCKS.RULE
Member:  TRUCKS.CLASSIFICATION.RULES
...

OwnSlot:  PREMISE
  Inheritance:  UNION
  ActiveValues:  WFFINDEX
  Values:  /Wff: (?X IS IN CLASS TRUCKS)
           /Wff: (THE WEIGHT OF ?X IS ?VAR29)
           /Wff: (GREATERP ?VAR29 10000)
           /Wff: (?X HAS AT LEAST 10 WHEELS)
           /Wff: (NOT (THE COLOR OF ?X IS RED))

------------------------------------
art._57

InstanceOf :  norms_for_indirect_transfer_of_reve-
nues_abroad
...

SubjectOfTheImposition  :  transnational_company
TerritorialValidity  :  france_
ValidityStart :
ValidityEnd :
DocumentationSource :  french_general_taxation_law
...

Procedure1 :
  topic  :  bloc-1
  premise  :  bloc-2
  norm  :  bloc-3
  exceptions  :
  commentaries  :
  ...

BLOC-1 : (ALTERN (COORD t1 t2 t3) (COORD t1 t4 t5))

t1) PRODUCE  SUBJ   x1
             OBJ    (SPECIF calculation_ income_tax )
             DEST   x2 : france_

    x1 = human_being_or_social_body ; x2 = company_

t2) OWN      SUBJ   x2 : france_
             OBJ    (SPECIF control_power x3 )

    x3 = company_ ; x2  ≠)(x3

t3) EXIST    SUBJ   x3 :foreign_country

t4) OWN      SUBJ   x3 : foreign_country
             OBJ    (SPECIF authority_ x2 )

t5) EXIST    SUBJ   x2 : france_

"determination of the income tax payable by companies
under the authority of companies domiciled abroad, or
which control such companies"
   ...
```

Figure 18. Two examples of the use of frame-like structures to implement rule-like structures. The first part of the figure concerns the use of a frame ("Unit" in KEE jargon) BIG.NON.RED.TRUCKS.RULE—specific term (Member) of the class TRUCK.CLASSIFICATION.RULES—to implement a rule for identifying big nonred trucks. Only the slot PREMISE is shown. The second part of the figure shows the NKRL representation of the beginning (the topic part of a complex rule) of a normative text (the rule) which corresponds to article n° 57 of the French "General Taxation Law": "In order to determine the income tax payable by companies which are under the authority of, or which exercise a control over, companies domiciled abroad . . .". art._57 (the global NKRL representation of the normative rule) is interpreted as an individual instance of the general concept *norms_for_indirect_transfer_of_revenues_abroad.*

KL-ONE includes a classifier that, on the basis of the subsumption relationships, automatically places new concepts into their correct place in the hierarchy.

Terminological languages generalize KL-ONE's ideas by operating, among other things, a very precise distinction between terminological (TBox) and assertional (ABox) knowledge. Terminological knowledge captures the intensional aspects of a domain. The domain representation is expressed, as in KL-ONE, in terms of concepts and roles. Concepts describe a set of notions of the domain, whereas the associated roles denote binary relations among concepts. A set of operators is provided, which allow defining complex concepts in terms of existing concepts and restrictions on roles. Assertional knowledge describes the extensional aspects of a domain and concerns the individuals constituting factual entities which are instances of the concepts proper to the terminological component. Considering an implemented terminological application as a knowledge base (KB), the TBox is the general schema of the KB that concerns the classes of individuals to be represented, their general properties, and mutual relationships. The ABox is a partial instantiation of this schema that contains assertions linking individuals with classes or individuals with each other.

For example, the description of an individual mary (assertional knowledge, ABox) in BACK is expressed as in Fig. 19 (93). The meaning of the different coding elements used in Fig. 19 is the following (here we use for the concepts, instances and roles = slots, the usual typographical conventions we have used until now):

- mary_ is the symbolic name which characterizes a unique individual in the knowledge base. mary_ is an instance of the concept *person_*.
- There is at most one different individual in the Child relation (role) with mary_. All individuals in the Child relation are instances of the concept *female_*.
- The Age of all the individuals in the Daughter relation with mary_ has a value greater than 10, and there is an individual named louise_ in the Child relation with mary_.
- *person_* and *female_* are concepts which must be defined by the user. Child, Daughter, and Age are roles which must also be defined by the user.
- **and, atmost, all, gt,** and **:** are built-in term-forming operators for building complex descriptions; see also Fig. 20 later. The term-forming operators introduce the roles associated with the concepts or individuals and the constraints linked with these roles. The constraints concern in general (1) the co-domain of the role (**all**), that is the concept which is the target of the relation established by the role; and (2) the cardinality of the role (**atmost, gt**),

```
mary_  ::  person_   and
           atmost (1,Child) and
           all(Child,female_)    and
           all(Daughter, all(Age, gt(10)))  and
           Child:louise_.
```

Figure 19. An example of individual entity (assertional knowledge, ABox) according to one of the best known terminological languages, BACK.

```
person_      :< anything_.
female_      :< not (male_).
Child        :< domain(person_) and range(person_).
Daughter     := Child and range(female_).
parent_      := atleast(1, Child).
mother_      := parent_ and female_.
grandmother_ := female_ and atleast(1,Child and range(parent_)).
```

Figure 20. Definition of concepts and roles in BACK. Concepts and roles can be primitive or defined. In this figure, :< and := are the operators for introducing, the first, primitive concepts and roles, and the second, defined concepts and roles.

that is the minimum and maximum number of elementary values that can be associated with the role. **::** is the built-in operator for associating individuals with their descriptions.

The definitions (descriptions) of the concepts and roles used in Fig. 19 (and of some related concepts and roles) are given in Fig. 20. Note that

- Concepts, in BACK as in KL-ONE and the other terminological languages, can be primitive concepts or defined concepts. The former are atomic (without definition), and are used in describing the latter. If a concept is defined, then it is linked with a description. Analogously, roles can be primitive or defined. In Fig. 20, :< and := are the operators for introducing, first, primitive concepts and roles, and second, defined concepts and roles. The features associated with a primitive concept are "necessary." Those associated with a defined concept are necessary and sufficient. The insertion of a defined concept in the concept hierarchy is achieved under the control of a classifier à la KL-ONE; see also later.
- *anything_* is the built-in universal concept, which is true for any individual. *nothing_* is the dual empty concept.
- The built-in operator **and** indicates generally that a concept (role) is defined as a conjunction of concepts (roles), which are the immediate ancestors of the new concept (role) in the hierarchy. Then the roles are also inserted in a hierarchical organization. See in Fig. 20 the role Daughter which is subsumed by Child. **atleast** is a built-in operator used to specify the cardinality of a role. **domain** and **range** are built-in operators for building role descriptions. **domain** specifies the sort of concept with which the role can be associated. **range** is the sort of concept that can fill the role.

Reasoning in BACK and in the other terminological languages includes at least the following operations: consistency checking, completion of partial descriptions and classification (95). Consistency checking involves coherence control in the definitions (descriptions) of concepts and the description of individuals. For example, the following constraint expression for a role: **atmost**(0, R) **and atleast**(1, R), is not admissible for any possible R, given that this role should be filled simultaneously by at least a value and at most zero values. Another example involves the definition of a concept where the role Child is filled by individuals that are, at the same time, instances of *male_* and *mother_*. A definition like this is not contradictory per se, but it is in contrast with the definition of *mother_* in Fig. 20, where *mother_* is defined as *female_*. Completion means being able to derive all of the consequences from the definition of the concepts, the descriptions of the individuals, and the application of all of the possible rules defined for a given terminological application. For example, querying a system that contains the description of Fig. 19 for all of the individuals older than 10 after having introduced the rule: **atleast**(1, Child) ⇒ **all**[Age, **gt**(13)], which states that the restriction of having at least a child implies that age must be greater than 13, allows retrieving both mary_ and louise_. The first value is retrieved because of the application of the rule, and the second because the constraint for Daughter in the description of mary_ is propagated to the filler of Child, given that Daughter, according to Fig. 20, must be a Child.

The last modality of reasoning is proper to the terminological languages, and involves the process of automatically finding the correct position of a concept in the hierarchy of all of the concepts. In particular, for each concept it is possible to find the more general ones, the most specific ones, and the disjoint ones. This process is based on the subsumption principle (see previous discussion). For example, according to the so-called "Normalization-Comparison" approach, subsumption can be determined by making syntactic comparisons between the defining structures of concepts C and D. After a normalization phase in which all the components of a description are developed and rearranged, the defined concepts are replaced by their definitions (in this way, all the symbols denote primitive roles and concepts). Now it becomes possible to compare two descriptions by executing relatively few operations, usually by comparing pairs of terms built with the same operator. Let us suppose a concept C whose defining description is: *game_* **and atleast**(2, Participant). Now suppose that we introduce a concept D defined as: *game_* **and atleast**(4, Participant) **and all**{Participant, [*person_* **and all**(Gender, *female_*)]}, that is a game with at least four participants where the fillers of the Participant role must be instances of the concept *person_*, which have themselves the Gender role filled with instances of the concept *female_*. Concept D will be subsumed by concept C.

Subsumption (and more generally terminological reasoning) is a very complex problem. This fact is intuitively evident when considering the complexity of the description that can be used to define the concepts and describe the individuals with respect, for example, to the relatively simple organization of the frame systems examined in the previous section. All of the proposed terminological languages mentioned before (with the exception of KRIS) are characterized by incomplete reasoning procedures. This means that some inferences are missing and that, in some cases, it is also impossible to identify precisely their semantic characteristics. For several systems, like LOOM, it is not even known if complete procedures can ever exist. From the point of view of computational complexity, a well-known result established first in Ref. 96 and confirmed by later research says, in very simple terms that subsumption is tractable (i.e., it is solvable in polynomial time in the worst case) for the simplest terminological languages, but it becomes intractable even for very slight extensions of these languages (e.g., when adding a term-forming like **restrict** to avoid the use of cumbersome combinations of **or, and,** and **not**). Moreover, it was proved undecidable for languages like KL-ONE and NIKL.

Note that these sorts of problems, even if they have been particularly studied in a terminological logic context (95) are absolutely general, and they involve all of the types of symbolic knowledge representation (KR) we have examined until now. For example, FOL has well-defined semantics and very strong deductive capabilities but, when its expressive power is extended to cope exactly with all of the relevant facts and entities of a given application domain, it becomes quickly computationally intractable, where intractability ranges from undecidability (i.e., the impossibility to determine whether one sentence follows from another) to NP-completeness (i.e., the impossibility of solving a problem in time polynomially proportional to the size of the problem description). Faced with this problem of the tractability of reasoning, all of the proposed approaches to symbolic KR lay between two extreme positions:

- The first considers only KR languages that have limited expressive power (accepting the risk that they could be of a limited practical interest for describing a certain number of domains) but that show tractable inferential capabilities. Following this approach, some terminological languages for example, KRYPTON and CLASSIC, supply limited tractable formalism for expressing concepts. Filling the gap between what can be expressed in the language and what is needed by a specific application is left to the user, who normally resorts to programs written in a procedural language.
- The second accepts, on the contrary, the fact that general-purpose symbolic KR languages are intractable or even undecidable, and then favors expressiveness with respect to the computational tractability. Note also that, from a practical point of view, problems about computational tractability normally concern only the worst cases. Then incomplete procedures are considered acceptable in terminological languages like NIKL, LOOM, and BACK.

To sum up, "There is a tradeoff between the expressiveness of a representational language and its computational tractability . . . We do believe, however, that the tradeoff discussed here is fundamental. As long as we are dealing with computational systems that reason automatically (without any special intervention or advice) and correctly (once we define what *that* means), we will be able to locate where they stand on the tradeoff: they will either be limited in what knowledge they can represent or unlimited in the reasoning effort they may require (43, pp. 42–43)."

KNOWLEDGE MANAGEMENT: SOME PRACTICAL ASPECTS

We conclude this article by describing briefly two specific applications of the representational principles examined in the previous sections, knowledge base management, and tools and support for knowledge management.

Knowledge Base Management

Knowledge base management systems (KBMSs), or intelligent database systems (IDBSs) (97), are characterized, in the first place, by the use of data models derived from AI research in the knowledge representational domain. We recall here that a data model, according to the database (DB) terminology, represents a logical organization of real-world objects (entities), of the constraints on them, and of their relationships. A DB system implements a data model. Then the use of AI data models implies the possibility, unfeasible in the traditional (relational) database management systems (DBMSs) of using advanced inferential techniques.

Also note that the most advanced KBMSs adopt (at least implicitly) architecture based on an organization in the expert systems (ESs) style, that is, composed of a fact database (FDB) and of a rule base (RLB). The FDB is concerned with the so-called persistency problem. In ordinary expert systems, data needed for an application are (normally) introduced by the user according to the system's requests (i.e., in small quantity and only when necessary). They reside in volatile memory and, therefore, they disappear as soon as the particular application is finished. The same happens to the intermediate results deduced in the course of the reasoning process. This cannot be accepted for the facts of a knowledge base. As with information in an ordinary DB, it must be possible to reuse them (i.e., facts must be permanently maintained independently of any application, even when the FDB is not being accessed). Because of this and its huge dimensions, normally the fact database of a KBMS cannot reside in volatile memory (central memory), but it must be organized on secondary memory (mass memory).

KBMSs (IDBSs) can be classified according to the knowledge representational technique used to encode their own knowledge. For example, deductive databases are based on the cooperation between (1) an intensional database corresponding to the RLB that contains logic formulas, that is, sets of assertions in PROLOG, DATALOG, etc. style, and (2) an extensional database corresponding to the FDB that contains base relations stored explicitly in the secondary storage (e.g., a relational DB). The aim is to apply the inferential mechanisms proper to the logic approach to the RLB formulas to derive, from base relations, information not explicitly stored in the FDB (virtual relations). The RLB rules can also be represented, obviously, as production rules, frames, or terminological logics statements. In this last case, the Tbox and the Abox are, respectively, good candidates for implementing the RLB and FDB. Many of these solutions, even if, at least potentially, very powerful from a deductive point of view, are characterized, however, by a low level of computational effectiveness. Very often knowledge bases and IDBSs are very small, given that their FDB resides only in central memory. In general, they provide only limited services for recovery, protection, integrity maintenance, concurrent access to distributed knowledge bases, etc., if any are provided at all.

Then a standard solution for realizing powerful KBMSs consists of coupling some sort of KBSs with (traditional) DBs. DBs are supposed to supply the KBSs with the correct quantity of data required to drive their inferencing mechanisms, while still preserving their basic functions (concurrency, etc.). This coupling has been realized by using all possible forms of association between DBs and KBSs, all sorts of loose or tight coupling. We limit ourselves to mentioning here the most popular type of coupled systems, the solutions involving coupling of traditional DBMSs with KESEs and ES shells where the two cooperating systems still preserve their autonomy. Normally (but not mandatory, see later), the KBS acts as a front end to be used as a repository for the domain-specific knowledge and to implement the reasoning mechanisms required for user tasks. Then the DBMS is used as a back end, containing facts required for front-end reasoning. An important point here is that, even when this distribution of duties is not exactly respected, an essential component of the overall system always consists of an already available, existing on-line database. Therefore, no need exists for restructuring and recoding the database information in-depth, nor for executing any unreasonable amount of change in existing applications. This advantage has been sometimes defined as the 80–20 rule (98). Using this type of approach for the setup of KBSs and IDBSs allows, at least in principle, achieving 80% of the benefits of integration at only 20% of the costs. Then, it is not surprising that, today, practically all existing commercial KBSs provide some (sometimes rudimentary) facilities to implement coupling with an existing DBMS. Note that architectural solutions very similar to these have also been developed in a logic programming context; see the so-called heterogeneous approach for constructing deductive databases.

Note that the attempt to couple a KBS and a DBMS while preserving their independence is not an easy task. Several authors (99,100), have noticed that there is a fundamental mismatch between the two types of subsystems that takes, at least, three different forms:

- The first involves the knowledge representational aspects (semantic mismatch). Simple relational algebra and "flat" relations proper to DBs are not always compatible with some advanced knowledge representational systems, for example, frames, which are used in many KESEs, to say nothing of the ad hoc ways of structuring the fact database which are current in many ES shells and which are realized only according to the constraints imposed by the characteristics of a particular inferential engine and by the properties of the problem at hand. Severe performance problems arise from this mismatch, often requiring, inter alia, the use of redundant data descriptions to make data exchange possible.
- A second type of mismatch involves the operational aspects of the global system (impedance mismatch). The inferential knowledge of an AI system, that is, the knowledge taking charge of the operational aspects of this system, is basically static because it is represented mainly by the declarative knowledge stored in the RLB. To the contrary, the operational component of a database system is dynamic and is represented by the knowledge embedded, in a procedural way, inside an application program. In complex applications, the data is retrieved from a DBMS using a DB query language, such as SQL, and then manipulated through routines written in a conventional programming language, such as C or PL/1. Cooperation between the two systems therefore implies, at least in principle, continuously translating static inferential processes into dynamic queries, and vice versa. Another aspect of this mismatch involves optimization. Although optimizing the KBS programs is left largely to the programmer, optimizing the relational DB is left to the system. Overall global optimization of computations is, at least in principle, precluded.
- A third type of mismatch involves the granularity of the data to be handled (granularity mismatch). An AI reasoning mechanism uses data to instantiate its variables. Therefore, it requires some data during each inference and under an atomic form (individual tuples of data values). To the contrary, a relational DBMS answers a query by returning results as sets of tuples. Accordingly, when the KBS breaks down a query into a sequence of queries on tuples, each of them incurs a heavy DBMS performance overhead. Therefore we lose the benefits of the set-oriented optimization characteristic of DBMSs. Moreover, unlike what happens with traditional algorithmic programming, it is impossible to completely anticipate the data access needs of a KBS given that, in these systems, control knowledge is separated from the (domain-specific) problem-solving knowledge, resulting in a reasoning process which is highly problem-dependent.

To realize the coupling, the five architectural solutions in Fig. 21 have been described in the literature (98).

Figure 21(a) corresponds to what we could call the "full bridge" solution. Coupling KBSs with existing DBMSs is realized here by explicitly building up a third component, an independent subsystem acting as a communication channel between the first two. No system dominates (at least in principle). This allows the DBMS to operate as a totally separate system with its own set of DB users. We note, however, that the appeal of this general solution is balanced by the practical difficulty of implementing efficient systems. All of the mismatches described previously come in fully. Figure 21(a) has at least two variants, depending on whether the control of the interactions between the two subsystems is located on the central bridge or distributed, as the processing, between the two original components. A simple solution in this category, which has been intensively used in the early times of the KBSs/DBMSs integration era, consists of using a flat file as the intermediary medium. To transfer information from the KBS to the database, the former writes the information into the file. This is transferred to the DBMS, which reads it and stores it as rows. The DBMS transfers information to the KBS by the converse procedure. Of course, this straightforward approach, which can be considered the prototype of any loose coupling approach, and which does not scale up well, is suitable only for knowledge-based applications that reason over a well-specified data set and where the interaction between the DBMS and the KBS is kept to a minimum.

An early, well-known full bridge solution is represented by the Dictionary Interface for Expert Systems and Databases (DIFEAD) system, implemented at Trinity College, Dublin, in the mideighties (101). DIFEAD is particularly interesting for at least two reasons:

- it is one of the first systems explicitly based on an architecture in the full-bridge style;
- it is among the first realizations that have grounded the functionalities of the KBS/DBMS interface (the KBSs are simple ESs in this case) on a concept proper to the database domain, the data dictionary concept.

We recall here that a data dictionary stores, in compiled format, both the different schemata (conceptual, etc.) which define the corresponding database and the rules assuring corre-

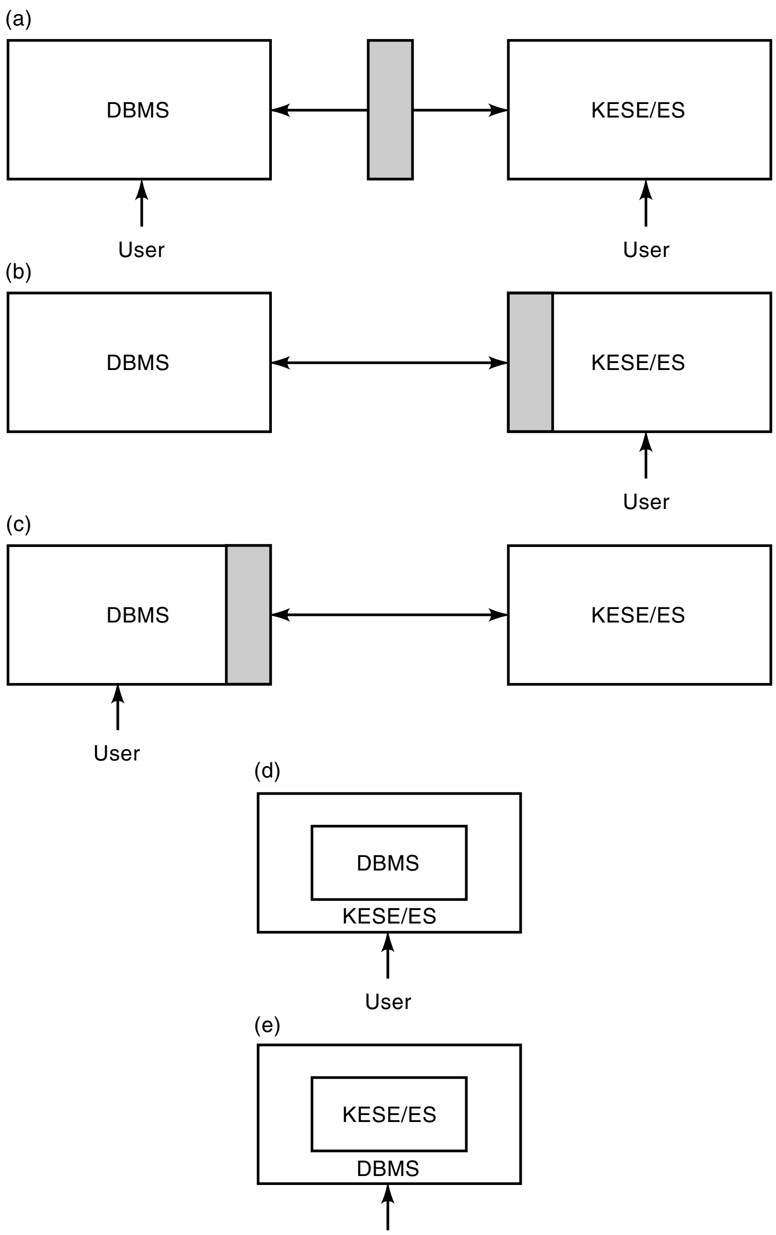

Figure 21. Architectural solutions proposed for coupling KBs (KESEs and ES shells) with (traditional) DBs. In these solutions, the two cooperating systems still preserve their autonomy. DBs are supposed to supply the KBSs with the correct quantity of data required to drive their inferencing mechanisms, while still preserving their basic functions (concurrency etc.). Normally (but not mandatory), the KBS acts as a front end to be used as a repository for the domain-specific knowledge and for implementing the reasoning mechanisms required for user tasks. Then the DBMS is used as a back end, containing facts required for front-end reasoning.

spondence among the different levels, along with a description of the meaning of the data. The dictionary itself may be conceived as a database. In this case, it fulfills the role of a metabase, that is, of a DB which describes the other DBs. In DIFEAD, the independent, central bridge module is called the metalevel component (MLC). It includes three main modules:

- The user interface module (UIM), whose main function is that of decoding a user request and sending it to the KBSs (ESs) via a second module, the metadata query module (MQM).
- The MQM is responsible for the communication between the ESs and the DB. In particular, it can decide whether an ES request can be answered automatically from the application DB or whether it is necessary to require input from the user. This choice is carried out by checking the Data Dictionary Directory System (DD/DS) that, in DIFEAD, is realized as a proper DBMS system (DIFEAD DBMS).
- The third module is the data update module (DUM) that is responsible for updating the DD/DS automatically after the ES has inferred a new fact.

The KBS systems included in DIFEAD were (relatively) simple systems based on the production rules paradigm (medical ESs). The reduction of the KBS to an ES can scale down greatly the semantic mismatch risk. The ⟨Attribute, Object, Value⟩ format (associative triples) often used for representing the terms in the rules of the simplest ES shells is, in fact, very compatible with the pairs attribute-value stored in a relational DB. When the conceptual model of the KBS is more complex, it becomes very difficult to implement a sort of general solution for the coupling that uses the full bridge approach.

To give only an idea of the difficulty of this task, we mention here the Advanced Information Management Systems (AIMS) project, developed in the framework of the ESPRIT programme of the European Communities. One of the AIMS' objective was integrating BACK (see the previous "terminological logics" section) with existing external relational DBs. A specific high-level language, Europe-Brücke (EB), was defined to implement an authentic full bridge between the two systems, that is, to allow an explicit and complete description of the links between the concepts in the BACK front end and the relations in the DB. The aim was that of realizing a wholly free connection, that is, to allow the possibility of mapping several relations on a single concept, of spreading a single relation on a concepts hierarchy, and of creating instances and attribute values as the result of complex queries.

Then the mapping information associated with a single BACK concept must provide a complex, two-level description of links between concept instances and DB relationships:

- Main data source links: databases and relations (tables) from which the keys of the tuples containing the instances description must be retrieved. The corresponding EB predicate is

  link(concept_name, tables_specification & **condition** (condition) & **name_from_backbase** (role_list))

- Fillers retrieval links: information about the DB links that must be followed to obtain the role fillers of the instances. DB links are expressed in terms of the involved DB tables, the relationships among the fields, and the conditions that must be satisfied by the tuples retrieved in this way. Then the corresponding EB predicate is

  link (role_name, **for_concept**(back_concept_name) & **with_range**(range_type) & **extract** (extract_statement) & **group_by** (field_name) & **option** (option_name))

The architectural solutions of Fig. 21(b) consist of extending a KBS with components proper to a DBMS. This implies normally implementing two sorts of modifications in the characteristic features of the KBS:

- The first consists of an extension with database functionalities of the AI language used in the KBS. For example, in a well-known product in Fig. 21(b) style distributed by Trinzic Corporation Limited, whose name is KBMS (sic), the left-hand side of the "if-then" rules includes support for ANDs and ORs, relations, algebraic expressions, parametric functions, and also a SQL-like dot qualification of attribute names to allow for multiple bindings of an object. Analogously, the right hand side includes commands like CREATE, UPDATE, DELETE, CALL, PRINT, RUN, etc. Similar solutions have been adopted in IntelliCorp's KEEConnection product, one of the first realizations in this domain. These allow KEE (see previous discussion) to access DB information, as if it were part of the KEE knowledge base, by automatically formulating SQL queries to move data between the DB (Fact DataBase) and the KB (Rule Base).
- The second type of modifications involves the inferential engine of the KBS, which now must be provided with data management functions allowing it to gain direct access to a generalized DBMS. The database functionalities of the inferential engine can be realized according to a loose or tight coupling approach. It must be noted that, especially when a tight coupling is chosen and in spite of the use of a general-purpose DBMS, the DB itself is completely devoted to the KBS application. This fact is symbolized, in Fig. 21(b), by indicating that the user gains access to the global system only through the KBS subsystem.

Because of the previous modifications which affect in-depth the normal characteristics of the KBS, Fig. 21(b) is less general (but more easy to implement) than that of Fig. 21(a). Moreover, the KBS subsystem can access the associated DB(s) only if the logical schema of this last component has been explicitly enclosed in the interface's structure, for example by using again some sort of data dictionary approach. This is why, until very recently, in the systems implementing the architecture of Fig. 21(b) each KBS/DB link provided access to a specific version of a DBMS product running under a specific operating system on a specific machine.

For example, in Trinzic's KBMS, the back end component is made up of external DBMSs (it is possible to access multiple DBs simultaneously). When a rule needs a data object, KBMS can automatically retrieve it through an automatic database interface. For example for applications requiring information from relational systems, KBMS automatically generates the SQL statement needed to access or update the data. If the data is moved to a different storage facility, no changes to the application rules are needed. Automatic database interfaces are based on the Automatic Data Definition (ADD) facility. ADD automatically loads data definition information from a data dictionary or catalogue into KBMS. ADDs are offered for the most popular DBMS and file storage systems: DB2, SQL/DS, IMS, CA-IDMS, VSAM, Adabas, Teradata, etc. (IBM mainframe environment); Rdb, ORACLE, RMS (DEC environment); OS/2 Database Manager (PC environment). KBMS also provides a manual DB interface, relying on user-written data access procedures and allows building applications that require access to data not automatically accessed by KBMS. Analogously, KEEConnection can connect with a fixed number of DBMSs using different network protocols, but only if these systems are anticipated in the product design.

Even if the basic implementing techniques of the solutions of Fig. 21(b) have not substantially changed with respect to the choices outlined before, the most modern realizations of this architecture try to be as general as possible, and solution 21(b) represents the architectural solution adopted by the vendors of the main ES tools to provide their systems with some elementary possibilities of extracting information from a database. To give only an example, see the 4.0 version of the EXSYS Professional tool (EXSYS Inc., Albuquerque, NM) which can now access up to 17 different SQL DBMSs.

Note, however, that

1. no standard approach exists for realizing the access functions, even if variants of the data dictionary technique are largely used;

2. the automatic behavior of the connection is often very rudimentary, and a lot of additional programming effort is often necessary to retrieve the data correctly from the DB.

The approach described by Fig. 21(c) is a symmetrical version of the previous one and consists of extending a DBMS with components proper to a KBS. When the database application must access the inferential engine and the knowledge base of the KBS subsystem, two strategies are usually employed, resulting in an explicit or implicit access procedure (102). In the first case, which uses a procedural call interface, an explicit call to the KBS must be inserted in the application program. This is the strategy followed in many of the commercial solutions (Cullinet. . .) to the integration problem. In the second case, the application itself does not explicitly call the KBS, and all access to the inferential engine is through the same query interface used to access data. Queries look like ordinary QSL queries without any explicit mention of a possible intervention of the KBS side. When some of the attributes mentioned in the query must be derived (i.e., their values are not explicitly stored in the DB), their values are obtained by inference from the KBS. Information about how to deal with such attributes is transparent to the user and stored in an active repository. For example in a query like "select amount, recommendation from credit approval where . . .", which refers to a credit authorization application (103, p. 28), the repository knows that amount and recommendation are derived attributes and triggers the corresponding rules in the rule base of the KBS. Note that all the architectural solutions in the Fig. 21(c) style can also be classed under the label "rule-based extensions" of the DBMSs and OODBMSs. For more technical details see, the "Database" articles in the Encyclopedia.

In the approaches described in Fig. 21(d) and 21(e), the functionalities of the DB and KBS systems are strongly integrated, and the designer is concerned with only one environment. This means that data model used in the DB component and the knowledge representational language of the KBS component are now unified. As a consequence, any possibility of semantic mismatch (see previous discussion) is avoided. Systems like these represent, however, a (at least partial) departure from the traditional approaches to integration. In the literature, descriptions of systems based on the solutions in Fig. 21(d) and 21(e) which are not simply general suggestions or, at best, experimental prototypes are, therefore, still relatively rare. We can add that commercial systems based on the architecture of Fig. 21d will probably constitute an exception in the future, at least from a strict KBS/DB integration point of view. When advanced semantic models (frames, objects, description logics . . .) are used to describe even the information in the FDB to achieve complete knowledge/data transparency, we obtain some unconventional (and controversial) pure AI systems in the style of TELOS (103), and CYC (104). From a standard KBS/DB point of view, the main disadvantage of solution 21(d) is that it requires constructing ex nihilo a DB system after (or during) the set up of the KBS. In many cases, this implies the need for long sessions of sequential dialogue with the user to collect the input data, whereas using the solution 21(e), the DB already contains the data needed to feed the KBS. Moreover, the DBMS technology is more stable and mature than the KBS technology, and the installed base of DBs is definitely larger than the KBSs base. A number of conventional applications already use the DBMS technology. Therefore, at least in a context of strong integration, DBs are probably a better place for incorporating KBS functionalities than vice versa.

We can mention two running systems using the Fig. 21(e) solution, American Red Cross Health Education System (ARCHES), which relatively simple, and the KBase system, which is more complex and is used to simulate the behavior of a scheduling expert in building construction (98).

Tools and Support for Knowledge Management

Before the mid-1970s, no real tool existed for facilitating the development of KBSs. These (ESs in the majority) were set up by writing directly large amounts of LISP or PROLOG code. LISP (and its various dialects) was the language of choice in the US. A beginning of normalization in the LISP field was reached in the seventies with the introduction of a LISP standard called COMMON LISP. In Europe and later in Japan (Fifth Generation Project), developers of KBSs preferred PROLOG. Given the complexity of these two languages, the necessity of building up the systems from scratch (with the consequence of experiencing very large development times) and the existence of few LISP and PROLOG programmers who worked mainly in an academic environment, very few KBSs (in the great majority in prototypical academic systems) were built up in the first twenty years of the existence of artificial intelligence (AI). We recall here that the official year of birth of AI is 1956 on the occasion of a famous Summer Workshop at Dartmouth College.

The mid-1970s' turning point resulted from the success of the MYCIN project. Developed in INTERLISP, a dialect of LISP, and including a knowledge base of about 500 production rules, this system was developed according to the architecture now considered the standard for developing ESs (see Fig. 8), which separates the proper knowledge base from the other modules of the system, working memory, inference engine, interfaces etc. MYCIN developers realized that, by suppressing the medical knowledge base of MYCIN, they could obtain an empty system, a shell, ready to be ported to other applications, based on inserting in the shell the knowledge base suited to the new application. Then the first shell, Essential MYCIN (EMYCIN) was born. One of the first utilizations of EMYCIN was the construction of PUFF, another ES in the medical field, where the rules of the knowledge base now related to diagnosing of pulmonary problems, instead of dealing with infectious blood diseases, as in MYCIN. In 1988, the percentage of ESs (more generally KBSs) developed by using a shell was already about 50%. About 25% of the applications still developed in pure LISP, and the rest were shared among PROLOG, OPS5, and other programming languages.

The first ES tools were strictly rule-based. Then a second revolution occurred in the mid-1980s, when the first environments for constructing complex frame-based KBSs arrived on the market. We have called these powerful environments knowledge engineering software environments (KESEs). KEE was introduced in 1983. ART by Inference Corp. was disclosed at the American Association for Artificial Intelligence (AAAI) Conference in Austin in the summer of 1985. Knowledge Craft by the Carnegie Group was commercialized later in the same year. For a good while KEE, ART, and Knowledge Craft have represented the inescapable trilogy of high-level tools for constructing the most powerful KBSs. Note that KBSs are not only ESs but also, to give only an example, complex computational linguistic applications.

An up-to-date review of the most important tools on the market today, arranged into seven classes, and an account of the criteria for selecting them can be found in Ref. 105. The first class of tools includes the pure AI languages, LISP, PROLOG (and, more recently, C and C++) and mainly OPS5 (OPS = Official Production Language), a rule-based programming language (see also the previous section "Production Rules") whose popularity is linked mainly with the success of the R1 (later called XCON) expert system, built up by John McDermott to help DEC configure VAX computer systems automatically (106). In the rule-based tools, tools basically following the EMYCIN philosophy, we can recall CLIPS, developed about 1985 at the NASA Johnson Space Center and freely available for a nominal fee. CLIPS adds procedural and object-oriented facilities to the basic production rules paradigm for knowledge representation. Other well-known tools in the rule-based class are Gensym's G2, Ilog's RULES, Teknowledge's M4, etc. In the frame-based tools class, KEE is now supplanted in practice by the new Intellicorp products, ProKappa and Kappa. C/C++ versions of ART and Knowledge Craft have been developed. The C/C++ version of Carnegie Representation Language (CRL), the frame-based knowledge representation language at the core of Knowledge Craft, is now called ROCK. A fourth class of tools includes the fuzzy logic tools. For example, FuzzyCLIPS 6.02 is a version of the CLIPS shell enhanced with extensions for representing and manipulating fuzzy facts and rules. Attar Software's XpertRule for Windows includes, with the usual Es tools, specific tools for fuzzy logic and genetic algorithm optimization. The next class of tools, induction tools, generates rules from examples and are the result of AI research in machine learning (see the article MACHINE LEARNING). An example is the free FOCL software tool, that learns Horn clause programs from examples and (optionally) background knowledge. The sixth class of tools is represented by the case-based reasoning (CBR) tools that use past experience (solved cases) to solve similar current problems (see the PLANNING articles). Well-known tools in this class are Inference Corporation's CBR Express, Haley's Easy Reasoner, Cognitive's ReMind, etc. The seventh and last class is associated with the so-called domain-specific tools, tools used for developing KBSs in a particular problem-solving domain (control, design, diagnosis, instruction, planning etc.). Some of them are commercially available, and others are developed by professional organizations for their specific needs. The tools of this class have gained particular importance in recent years.

Reference 105 contains a series of diagrams that summarize visually the major trends in the KBS tools market in recent years. For example, it can be seen that annual global sales of development tools decreased between 1990 and 1991, as a final consequence of the "AI winter" of the second half of the 1980s. After 1992, they rebounded quite well, as proof of the renewed confidence of the software industry in AI possibilities, and sales were about $200 millions in 1995. The diagrams also show a very sharp decline in annual sales of pure AI languages tools (mainly LISP tools) after 1990, and an outstanding increase in sales of domain-specific tools. A final interesting phenomenon involves the decline, after 1990, in the sale of tools specifically configured for PC and Macintoshes, as well as a decrease in the sale of the mainframe tools. Sales of workstation tools have, to the contrary, increased each year since 1988, and this trend will probably continue.

BIBLIOGRAPHY

1. A. Newell, The knowledge level, *Artif. Intell.,* **18**: 87–127, 1982.
2. G. Schreiber, B. Wielinga, and J. Breuker, *KADS: A Principled Approach in Knowledge-Based System Developments,* London: Academic Press, 1993.
3. J. Breuker and W. van de Velde (eds.), *Common KADS Library for Expertise Modeling,* Amsterdam: IOS Press, 1994.
4. L. Steels, The componential framework and its role in reusability, in J.-M. David and J.-P. Krivine (eds.), *Second Generation Expert Systems,* Berlin: Springer-Verlag, 1993.
5. H. Eriksson et al., Task modeling with reusable problem-solving methods, *Artif. Intell.,* **79**: 293–326, 1995.
6. N. F. Noy and C. D. Hafner, The state of the art in ontology design-A survey and comparative review, *AI Mag.,* **18** (3): 53–74, 1997.
7. B. Chandrasekaran, Design problem solving: A task analysis, *AI Mag.,* **11** (4): 59–71, 1990.
8. R. Neches et al., Enabling technology for knowledge sharing, *AI Mag.,* **12** (3): 36–56, 1991.
9. M. R. Genesereth and R. E. Fikes (eds.), *Knowledge Interchange Format-Version 3.0 Reference Manual* (Report Logic-92-1). Stanford, CA: Computer Science Dept. of Stanford Univ., 1992.
10. G. van Heijst, R. van der Spek, and E. Kruizinga, Organizing corporate memories, *Proc. 10th Banff Knowledge Acquisition Knowledge-Based Syst. Workshop,* 1996.
11. E. A. Fegenbaum, Knowledge engineering: The applied side of artificial intelligence, *Ann. NY Acad. Sci.,* **246**: 91–107, 1984.
12. R. J. Brachman and H. J. Levesque (eds.), *Readings in Knowledge Representation,* San Francisco: Morgan Kaufmann, 1985.
13. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry,* Cambridge, MA: MIT Press, 1969.
14. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning internal representations by error propagation, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition,* Cambridge, MA: MIT Press, 1986.
15. P. D. Wasserman, *Neural Computing: Theory and Practice,* New York: Van Nostrand Reinhold, 1989.
16. G. Josin, Integrating neural networks with robots, *AI Expert,* **3** (8): 50–58, 1988.
17. J. H. Holland, *Adaptation in Natural and Artificial Systems,* Ann Arbor: Univ. Michigan Press, 1975.
18. W. M. Spears et al., An overview of evolutionary computation, *Proc. 1993 Eur. Conf. Mach. Learning,* 1993, ECML-93.
19. J. Koza, *Genetic Programming,* Cambridge, MA: MIT Press, 1992.
20. L. J. Eshelman, R. Caruna, and J. D. Schaffer, Biases in the crossover landscape, *Proc. 3rd Int. Conf. Genet. Algorithms,* 1989.
21. G. Syswerda, Uniform crossover in genetic algorithms, *Proc. 3rd Int. Conf. Genet. Algorithms,* 1989.
22. D. Beasley, D. R. Bull, and R. R. Martin, An overview of genetic algorithms: Part 1, Fundamentals, *Univ. Comput.,* **15** (2): 58–69, 1993.
23. D. E. Goldberg, *Genetic Algorithms in Search Optimization, and Machine Learning,* Reading, MA: Addison–Wesley, 1989.

24. D. Beasley, D. R. Bull, and R. R. Martin, An overview of genetic algorithms: Part 2, research topics, *Univ. Comput.,* **15** (4): 170–181, 1993.
25. D. Whitley, *A Genetic Algorithm Tutorial* Technical Report CS-93-103, Fort Collins, CO, Computer Science Dept. Colorado State Univ., 1993.
26. L. A. Zadeh, Fuzzy sets, *Inf. Control,* **8**: 338–353, 1965.
27. B. Kosko, *Neural Networks and Fuzzy Systems—A Dynamical Systems Approach to Machine Intelligence,* Englewood Cliffs, NJ: Prentice-Hall, 1992.
28. H. J. Zimmermann, *Fuzzy Set Theory and its Applications,* 2nd ed., Norwell, MA: Kluwer, 1991.
29. G. Viot, Fuzzy logic: Concepts to constructs, *AI Expert,* **8** (11): 26–33, 1993.
30. I. B. Türksen, Hybrid systems: Fuzzy neural integration, in J. Liebowitz (ed.), *The Handbook of Applied Expert Systems,* Boca Raton, FL: CRC Press, 1998.
31. C.-C. Hung, Building a neuro-fuzzy learning control system, *AI Expert,* **8** (11): 40–49, 1993.
32. A. Newell and H. A. Simon, Computer science as empirical inquiry: Symbols and search, *Commun. ACM,* **19**: 113–126, 1976.
33. F. Lehmann (ed.), *Semantic Networks in Artificial Intelligence,* Oxford, UK: Pergamon Press, 1992.
34. J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine,* Reading, MA: Addison-Wesley, 1984.
35. G. P. Zarri, NKRL, a knowledge representation tool for encoding the 'meaning' of complex narrative texts, *Natural Language Eng.—Special Issue on Knowledge Representation Natural Language Process. Implemented Syst.,* **3**: 231–253, 1997.
36. J. A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM,* **12**: 23–41, 1965.
37. J. A. Robinson, Expressing expertise through logic programming, in D. Michie (ed.), *Introductory Readings in Expert Syst.,* New York: Gordon and Breach, 1982.
38. N. J. Nilsson, *Principles of Artificial Intelligence,* Palo Alto, CA: Tioga, 1980.
39. K. Knight, Unification: A multidisciplinary survey, *ACM Comput. Surv.,* **21**: 93–124, 1989.
40. R. A. Kowalski, Logic as a computer language, in K. L. Clark and S.-A. Tärnlund (eds.), *Logic Programming,* New York: Academic Press, 1982.
41. J. W. Lloyd, *Foundations of Logic Programming,* 2nd ed., New York: Springer-Verlag, 1987.
42. R. A. Kowalski, Algorithm = logic + control, *Commun. ACM,* **22**: 424–436, 1979.
43. R. J. Brachman and H. J. Levesque, A fundamental tradeoff in knowledge representation and reasoning, in R. J. Brachman and H. J. Levesque (eds.), *Readings in Knowledge Representation and Reasoning,* San Francisco: Morgan Kaufmann, 1985.
44. W. P. Dowling and J. H. Gallier, Linear-time algorithms for testing the satisfiability of propositional Horn formulae, *J. Logic Programming,* **1**: 267–284, 1984.
45. A. Colmerauer et al., *Un système de communication Homme-machine en Français* (Rapport de recherche). Marseille: Groupe d'Intelligence Artificielle de l'Université Aix-Marseille II, 1973.
46. P. Roussel, *PROLOG, Manuel de référence et d'utilisation.* Luminy: Groupe d'Intelligence Artificielle de l'Université Aix-Marseille II, 1975.
47. M. H. van Emden and R. A. Kowalski, The semantics of predicate logic as a programming language, *J. ACM,* **23**: 733–742, 1976.
48. A. Colmeraurer, PROLOG and Infinite Trees, in K. L. Clark and S.-A. Tärnlund (eds.), *Logic Programming,* London: Academic Press, 1982.
49. W. F. Clocksin and C. S. Mellish, *Programming in PROLOG,* Berlin: Springer-Verlag, 1981.
50. H. Gallaire and C. Lasserre, Metalevel control for logic programming, in K. L. Clark and S.-A. Tärnlund (eds.), *Logic Programming,* London: Academic Press, 1982.
51. E. L. Post, Formal reductions of the general combinatorial decision problem, *Amer. J. Math.,* **65**: 197–268, 1943.
52. A. Markov, *Theory of Algorithms,* Moscow: USSR National Academy of Sciences, 1954.
53. N. Chomsy, *Syntactic Structures,* The Hague: Mouton, 1957.
54. A. I. Vermesan, Foundation and application of expert system verification and validation, in J. Liebowitz (ed.), *The Handbook of Applied Expert System,* Boca Raton, FL: CRC Press LCC, 1998.
55. E. H. Shortliffe, *Computer-Based Medical Consultations: MYCIN,* New York: American-Elsevier, 1976.
56. P. Norvig, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp,* San Francisco: Morgan Kaufmann, 1992.
57. G. Shafer, *A Mathematical Theory of Evidence,* Princeton, NJ: Princeton Univ. Press, 1976.
58. P. R. Harrison and J. G. Kovalchik, Expert Systems and Uncertainty, in J. Liebowitz (ed.), *The Handbook of Applied Expert Systems,* Boca Raton, FL: CRC Press, 1998.
59. C. L. Forgy, Rete: A fast algorithm for the many pattern/many object match problem, *Artif. Intell.,* **19**: 17–37, 1982.
60. C. L. Forgy, The OPS languages: A historical overview, *PC AI,* **9** (5): 16–21, 1995.
61. P. Graham, Using the RETE algorithm, *AI Expert,* **5** (12): 46–51, 1990.
62. N. Fridman Noy and C. D. Hafner, The state of the art in ontology design-A survey and comparative review, *AI Mag.,* **18** (3): 53–74, 1997.
63. A. Gomez-Perez, Knowledge sharing and reuse, in J. Liebowitz (ed.), *The Handbook of Applied Expert Systems,* Boca Raton, FL: CRC Press LCC, 1998.
64. R. J. Brachman, What IS-A is and isn't: An analysis of taxonomic links in semantic network, *IEEE Comput.,* **16** (10): 30–36, 1983.
65. R. Fikes and T. Kehler, The role of frame-based representations in reasoning, *Commun. ACM,* **28**: 904–920, 1985.
66. D. B. Lenat et al., CYC: Toward programs with common sense, *Commun. ACM,* **33** (8): 30–49, 1990.
67. W. A. Woods, What's in a link: Foundations for semantic network, in D. G. Bobrow and A. M. Collins (eds.), *Representation and Understanding: Studies in Cognitive Sciences,* New York: Academic Press, 1975.
68. M. Minsky, A framework for representing knowledge, in P. H. Winston (ed.), *The Psychology of Computer Vision,* New York: McGraw-Hill, 1975.
69. R. J. Brachman, 'I lied about the trees' or, defaults and definitions in knowledge representation, *AI Mag.,* **6** (3): 80–93, 1985a.
70. D. W. Etherington, Formalizing nonmonotonic reasoning systems, *Artif. Intell.,* **31**: 41–85, 1987.
71. D. W. Etherington, *Reasoning with Incomplete Information.* San Francisco: Morgan Kaufmann, 1988.
72. R. Nado and R. Fikes, Saying more with frames: Slots as classes, in F. Lehmann (ed.), *Semantic Networks in Artificial Intelligence,* Oxford, UK: Pergamon Press, 1992.
73. R. Reiter, A logic for default reasoning, *Artif. Intell.,* **13**: 81–182, 1980.

74. R. Reiter and G. Criscuolo, Some representational issues in default reasoning, in N. J. Cercone (ed.), *Computational Linguistics,* Oxford, UK: Pergamon Press, 1983.

75. S. E. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge,* Cambridge, MA: MIT Press, 1979.

76. D. S. Touretzky, *The Mathematics of Inheritance Systems,* London: Pitman, 1986.

77. E. Sandewall, Non-monotonic inference rules for multiple inheritance with exceptions, *Proc. IEEE,* **74**: 1345–1353, 1986.

78. L. Padgham, Defeasable inheritance: A lattice based approach, in F. Lehmann (ed.), *Semantic Networks in Artificial Intelligence,* Oxford, UK: Pergamon Press, 1992.

79. P. J. Hayes, The logic of frames, in D. Metzing (ed.), *Frame Conceptions and Text Understanding,* Berlin: de Gruyter, 1979.

80. G. Ringland, Structured object representation—Schemata and frames, in G. A. Ringland and D. A. Duce (eds.), *Approaches to Knowledge Representation: An Introduction,* Letchworth: Research Studies Press, 1988.

81. D. Skuce, Conventions for reaching agreement on shared ontologies, *Proc. 9th Banff Knowledge Acquisition for Knowledge-Based Syst. Workshop,* 1995.

82. R. Wilensky, *Some Problems and Proposals for Knowledge Representation* (UCB/CSD Report n° 87/351), Berkeley, CA: University of California Computer Science Division, 1987.

83. U. Schiel, Abstractions in semantic networks: Axiom schemata for generalization, aggregation and grouping, *ACM Sigart Newsl.,* (107): 25–26, 1989.

84. M. E. Winston, R. Chaffin, and D. Herrmann, A taxonomy of part-whole relations, *Cognitive Sci.,* **11**: 417–444, 1987.

85. A. Artale et al., Part-whole relations in object-centered systems: An overview, *Data Knowledge Eng.,* **20**: 347–383, 1996.

86. G. P. Zarri, Semantic modeling of the content of (normative) natural language documents, in *Actes des dizèmes journées int. d'Avignon 'Les systèmes experts et leurs applications'-Conf. spécialisée sur le traitement du langage naturel,* Paris, 1992.

87. G. P. Zarri, Knowledge acquisition from complex narrative texts using the NKRL technology, *Proc. 9th Banff Knowledge Acquisition for Knowledge-Based Syst. Workshop,* 1995.

88. R. J. Brachman, V. Pigman Gilbert, and H. J. Levesque, An essential hybrid reasoning system: Knowledge and symbol level accounts in KRYPTON, *Proc. 9th Int. Joint Conf. Artif. Intell.,* San Francisco, 1985.

89. T. S. Kaczmarek, R. Bates, and G. Robins, Recent developments in NIKL, *Proc. 5th Natl. Conf. Artif. Intell.,* 1986.

90. R. MacGregor and R. Bates, *The LOOM Knowledge Representation Language* (Technical Report ISI/RS-87-188). Marina del Rey, CA: USC/Information Science Institute, 1987.

91. R. J. Brachman et al., Living with CLASSIC: When and how to use a KL-ONE-Like language, in J. F. Sowa (ed.), *Principles of Semantic Networks,* San Francisco: Morgan Kaufmann, 1991.

92. F. Baader and B. Hollunder, A terminological knowledge representation system with complete inference algorithm, *Proc. Workshop on Processing Declarative Knowledge,* 1991.

93. T. Hoppe et al., *BACK V5 Tutorial and Manual* (KIT Report 100). Berlin: Department of Computer Science of the Technische Universität, 1993.

94. R. J. Brachman and J. G. Schmolze, An overview of the KL-ONE knowledge representation system, *Cognitive Sci.,* **9**: 171–216, 1985.

95. M. Buchheit, F. M. Donini, and A. Schaerf, Decidable reasoning in terminological knowledge representation systems, *J. Artif. Intell. Res.,* **1**: 109–138, 1993.

96. H. J. Levesque, Foundations of a functional approach to knowledge representation, *Artif. Intell.,* **23**: 155–212, 1984.

97. E. Bertino and G. P. Zarri, *Intelligent Database Systems,* London: Addison-Wesley, in press.

98. C.-K. Soh, A.-K. Soh, and K.-Y. Lai, An approach to embed knowledge in database systems, *Eng. Appl. Artif. Intell.,* **5**: 413–423, 1992.

99. S. Tsur, LDL-A technology for the realization of tightly coupled expert database systems, *IEEE Expert,* **3** (3): 41–51, 1988.

100. S. Khoshafian, Modeling with object-oriented databases, *AI Expert,* **6** (10): 27–33, 1991.

101. A. Al-Zobaidie and J. B. Grimson, Expert systems and database systems: How can they serve each other?, *Expert Syst.,* **4** (1): 30–37, 1987.

102. B. Cohen, Merging expert systems and databases, *AI Expert,* **4** (2): 22–31, 1989.

103. J. Mylopoulos et al., TELOS: Representing knowledge about information systems, *ACM Trans. Inf. Syst.,* **8**: 325–362, 1990.

104. D. B. Lenat and R. V. Guha, *Building Large Knowledge Based Systems,* Reading, MA: Addison-Wesley, 1990.

105. J. Durkin, Expert System Development Tools, in J. Liebowitz (ed.), *The Handbook of Applied Expert Systems,* Boca Raton, FL: CRC Press, 1998.

106. J. McDermott, R1: The formative years, *AI Mag.,* **2** (2): 21–29, 1981.

GIAN PIERO ZARRI
CNRS-CAMS