# Computers for Symbolic Processing

BENJAMIN W. WAH, SENIOR MEMBER, IEEE, MATTHEW B. LOWRIE, AND GUO-JIE LI

*Invited Paper*

*In this paper, we provide a detailed survey on the motivations, design, applications, current status, and limitations of computers designed for symbolic processing. Symbolic processing applications are computations that are performed at the word, relation, or meaning levels. A major difference between symbolic and conventional numeric applications is that the knowledge used in symbolic applications may be fuzzy, uncertain, indeterminate, and ill represented. As a result, the collection, representation, and management of knowledge is more difficult in symbolic applications than in conventional numeric applications. We survey various techniques for knowledge representation and processing, from both the designers' and users' points of view. The design and choice of a suitable language for symbolic processing and the mapping of applications into a software architecture are then presented. We examine the design process of refining the application requirements into hardware and software architectures and discuss state-of-the-art sequential and parallel computers designed for symbolic processing.*

## I. Introduction

The development of the programming language IPL in the 1950s by Newell, Shaw, and Simon was a pioneering effort on symbolic processing by computers [172]. Data structures of unpredictable shape and size could be manipulated conveniently by programs written in IPL. Many of the early symbolic programs, including the Logic Theorist and the General Problem Solver, were written in IPL. The invention of Lisp in 1958 by John McCarthy further enhanced some of the programming tasks for symbolic processing. The language featured the use of conditional expressions recursively, representation of symbolic information externally by lists and internally by linked lists, and representation of program and data using the same data structures [152].

Recent advancements in applications of computers suggest that the processing of symbols rather than numbers will be the basis for the next generation of computers. This is highlighted by the numerous research efforts in Japan, Europe, and the United States [1], [196], [228]. Symbolic processing has been applied in a wide spectrum of areas; among them are pattern recognition, natural language processing, speech understanding, theorem proving, robotics, computer vision, and expert systems. Researchers in artificial intelligence, database, programming languages, cognitive science, psychology, and many others have addressed overlapping issues within the area of symbolic processing.

Conventional computers have been designed with tremendous numeric processing power as their focus, rather than symbolic processing power. The disparity between symbolic and numeric operations, therefore, calls for different architectures for symbolic processing and innovative research in computers for symbolic processing. A review of the state of the art in computers for symbolic processing is presented in this paper. The discussion proceeds in a top-down fashion. The relevant features and characteristics of symbolic processing are first presented. A perspective on the role of techniques and methodologies involved in the design process are discussed. Hardware and software architectures in different levels of design are classified. The general view of computers designed and/or used for a symbolic processing application is depicted in Fig. 1.

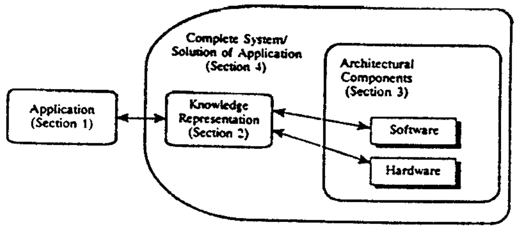
Complete System/ Solution of Application (Section 4)
Application (Section 1)
Knowledge Representation (Section 2)
Architectural Components (Section 3)
Software
Hardware

Fig. 1. Overview of this paper.

The section discussing each portion is indicated in the figure.

In Section I, a classification of general computations is developed. From this classification, a definition of symbolic processing is derived in Section I-A. Typical symbolic processing applications and their characteristics are discussed in Section I-B.

Manuscript received December 29, 1987; revised December 14, 1988. This work was supported in part by National Aeronautics and Space Administration Grant NCC 2-481 and National Science Foundation Grant MIP 85-19649. The research of M. B. Lowrie was also supported by a Ph.D. scholarship from AT&T Bell Laboratories.

B. W. Wah and M. B. Lowrie are with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana, IL 61801, USA.

G.-J. Li is with the Institute of Computing Technology, Academia Sinica, P.O. Box 2704-1, Beijing, People's Republic of China.

IEEE Log Number 8926701.

Knowledge representation and knowledge processing are two important characteristics of solutions to a symbolic processing problem. Knowledge representation refers to the technique for representing data and information in a computer and is discussed in Section II-A. Knowledge processing refers to the technique for controlling the manipulation of knowledge in the system and is the topic of Section II-B.

The design of a computer relies on various concepts and strategies for implementing knowledge processing techniques. Section III emphasizes the architectural concepts behind the design of symbolic processing systems. Software architectures are covered in Section III-A, and hardware architectures are studied in Section III-B.

A complete system for symbolic computation is the result of the application of design philosophy, architectural components, and available technology. Complete systems are the topic of Section IV. The status of many existing and experimental systems are discussed and compared.

Future symbolic processing systems will evolve as new concepts and technologies develop. Section V outlines some recent research that is likely to impact the design of symbolic processing systems in the future.

### *A. Classification of Computations*

One of the fundamental debates on intelligent behavior has been related to the explanation of what symbols are. A number of scientists view human beings and computers as physical symbolic systems that produce through time an evolving collection of symbolic structures. In their 1975 Turing award lecture, A. Newell and H. Simon stated a general scientific hypothesis—The Physical Symbol System Hypothesis [174]:

> "A physical symbol system has the necessary and sufficient means for general intelligent action."

By "necessary," they mean that any system that exhibits general intelligence will prove upon analysis to be a physical symbol system. By "sufficient," they mean that any physical symbol system of sufficient size can be organized further to exhibit general intelligence. Research on artificial intelligence (AI) addresses the sufficiency of physical symbol system for producing intelligent action, while investigators in cognitive psychology attempt to demonstrate the necessity of having a physical symbol system wherever intelligence is exhibited. Although empirical in nature, the continuous accumulation of empirical evidence on the above hypothesis in the last 30 years has formed the basis of much research on AI and cognitive science.

Since our focus is on computers for symbolic processing, we will first classify computations performed on computers. The definition of symbolic processing used in this paper is derived from this classification. There are five classes of computations: analog, numeric, word, relational, and meaning. *These classes are based on the primary unit of storage in the computation.*

*Analog:* The analog class of computation encompasses those computations that have continuous variables as the parameters of the functions it performs. This is not the primary area of computation in a digital computer, as digital computers use digital memory. In the computations discussed in the context of digital computers, this layer of computation primarily entails the measurements of parameters from the environment.

*Numeric:* In this class of computation, the primary unit upon which functions are performed represents magnitude. Many applications of computers fall into this category; functions on memory elements containing integers, floating point numbers, ... are numeric.

*Word:* In this class of computation, the parameters of functions are words that do not necessarily have quantitative value. Text processing is such an example.

*Relational:* In relational computations, functions operate on relations among words; that is, the primary unit of storage to be operated on are groups of words that have some relational interpretation.

*Meaning:* Very little research has been done on techniques for automated computation at the meaning level. The primary unit of evaluation is an interrelated series of relations that represent semantics and meaning.

A few examples to illustrate the various classes of computations are shown in Fig. 2. In Fig. 2(a), a standard super-

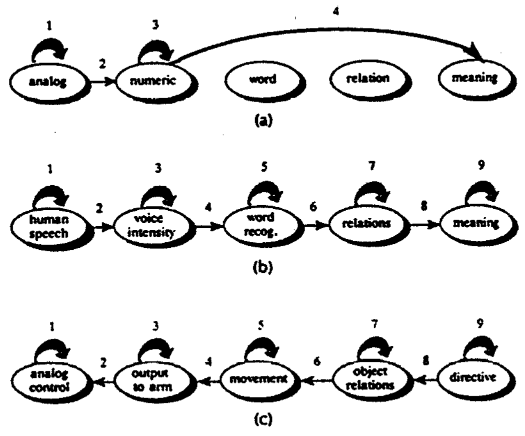

Fig. 2. Examples to illustrate model of symbolic processing applications. (a) Model of weather forecasting on a supercomputer. (b) Model of speech understanding. (c) Model of robot control.

computer application, weather forecasting, is presented. The computation begins with analog measurements of the atmosphere (arc 1). These measurements are then converted to numeric entities (arc 2). The majority of computation occurs in the numeric stage (arc 3, representing the conversion of numbers into different sets of numbers), with conversion to meaning done at the very end (arc 4, which may be done by humans instead of the computer). An example that uses the full spectrum of the classes of computations would be story comprehension from speech input. The flow of data would appear as in Fig. 2(b). Computation may also flow in the opposite direction. Robot control is an example of this, as depicted in Fig. 2(c).

The design of a computer system can be viewed as a problem in which performance with respect to the problem to be solved is to be maximized subject to cost constraints. Computations in more abstract classes are usually carried out by transformations into computations in more definite classes. For example, database queries function primarily at the word and relation level of computation. However, if we wish to know the average salary in a database of employees, numeric computations would be required. The design of this system may not, however, benefit from the inclusion of fast arithmetic units, because the performance gain may not be large enough to counteract the loss of other features that are necessitated by the cost constraint. The difference between this database system and a system that would include fast arithmetic units is in the emphasis of the computations.

A computer that is aimed at functioning at a more abstract level of computations should be able to perform computations that are found at a more definite level of computations. For instance, the database computer above should be able to perform fluid dynamics calculations. When this is done, however, the efficiency may be much poorer than a computer of comparable cost that is aimed at numeric calculations.

Using this classification, a concise description of what is meant by "symbolic processing" is possible. *Symbolic processing is defined as those computations that are performed at the same or more abstract level than the word level.* We call computers that are designed with specific orientation to carry out operations at or more abstract than the word level *symbolic processing computers.*

## *B. Characteristics of Symbolic Processing Applications*

In this section, symbolic processing applications and their overall features are presented. A few applications and their characteristics are presented in Table 1. The features that characterize general symbolic processing applications are listed below [239]. These features do not always apply to every instantiation of a symbolic processing problem and are intended as design guidelines that target general symbolic processing techniques.

- *Incomplete knowledge:* Many applications are *nondeterministic;* it is not possible to predict the flow of computation in advance. This is due to incomplete knowledge and understanding of the application. This lack of complete knowledge may also lead to *dynamic execution,* which refers to the possibility of new data structures and functions being created during the actual solution of the problem. In addition, data structures used in solution of the problem may be arbitrarily large, thereby necessitating dynamic allocation of memory, tasks, and other resources. System design should meet the need to cope with dynamic and nondeterministic execution. An architecture that can adapt to more efficiently perform computations not anticipated at design time is referred to as an *open system.*
- *Knowledge processing:* A computation can be viewed as manipulations on a set of data. In Section I-A, computations are classified on the basis of the nature of the data operated on. The nature of the operations performed on the data depends on the application and the storage technique for the data. Fig. 3 shows a possible view of a computation and illustrates that it is processing knowledge about the computation, whether that be algorithms, techniques for evaluation, or some other reasoning technique, that controls and dictates the manipulation of data. At a level above this, *meta-knowledge* includes the extent and origin of the domain knowledge of a particular object, the reliability of certain information, and the possibility that an event will occur. In other words, meta-knowledge is knowl-

Table 1 Some Symbolic Processing Applications

| Application | Characteristics |
|---|---|
| Problem solving general specific • programming/compilation • text processing • human interface | User inputs problem, system attempts to solve; user encodes solution of problem; meta-knowledge for specific problems is well understood; small-gain parallelism is predominant |
| Database management • variety of applications • often an integral part of larger systems | Organization of information for retrieval; efficient algorithms to consider all run-time possibilities are too complex; meta-knowledge is application dependent; large potential for parallelism, both small-grain and large-grain |
| Expert systems diagnosis • medical • plant disease • computer system errors design assistance • architecture • computer architecture • computer chips personal systems • business • finance • wine tasting others | Ill-structured collection of facts, inferences . . . as knowledge-intensive program in specific domain [96]; knowledge and meta-knowledge are usually provided by designers; large potential for parallelism |
| Natural language processing Understanding Generation Translation | Translate natural language to machine representation; translate machine representation to natural language; translate between two forms of natural language [2], [13], [229] |
| Computer vision signal processing pattern recognition image understanding | Primarily numeric at the signal-processing level; patterns viewed as sentences—symbolic in nature; higher-level reasoning at the image-understanding level [3], [76] |
| Learning experimentation deduction knowledge acquisition | Ability to adapt to environment to improve system efficiency; fundamental to symbolic processing [157], [191] |

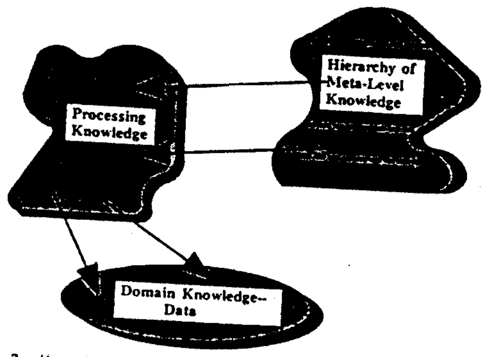

Fig. 3. Knowledge processing.

edge about knowledge [7], [49], [80]. Meta-level knowledge can be considered to exist in a single level or in a hierarchy [25]. In fact, there can be an arbitrary number of levels, each serving to direct the use of knowledge at the lower levels.

Meta-knowledge can be classified as deterministic or statistical according to correctness and performance considerations [39]. Deterministic meta-knowledge refers to the knowledge about precedence relationships, which results from a better understanding of the problem and helps to reduce the resource and time complexity. Statistical meta-knowledge can be used to order object-level actions in advance for efficient operations.

Rather than adding more heuristics to improve performance, more meta-knowledge about effective use of existing heuristics can be collected and developed. Meta-knowledge can also account for the formalization of belief, default reasoning, inference in changing situations, and others [7].

- *Symbolic primitives:* A general symbolic application may contain primitive symbolic operations. Typical operations are comparison, sorting, selection, matching, and logical operations such as union, negation and intersection, transitive closure, pattern retrieval and recognition. These operations may be performed at more than one level of computation (such as word or relation). Higher levels of computation may also contain complicated "primitive" operations such as unification.
- *Parallel and distributed processing:* Many symbolic applications exhibit a large potential for parallelism. Parallelism may be categorized into AND-parallelism and OR-parallelism. In AND-parallelism, a set of necessary and independent tasks are executed concurrently. OR-parallelism is a technique used to shorten the processing time in nondeterministic computations by evaluating alternatives at a decision point simultaneously.

## II. Symbolic Processing

A symbolic application is a problem in which the inputs and outputs are symbolic. *Symbolic processing* refers to the techniques employed by the system for finding the solutions of the application. The characteristics of symbolic applications have been discussed in the last section. The emphasis of this section is on the *features of symbolic processing* as they relate to the design of computers. In Section II-A, techniques for representing knowledge are discussed. The issues involved in the control of knowledge processing are presented in Section II-B.

### A. Knowledge Representation

In order to design an efficient computer for a given application, it is necessary to characterize the programs that will run on the computer. A primary decision in the solution of symbolic processing problems is the knowledge representation to be used [120], [149]. The issues involved include the selection of the appropriate symbolic structures to represent knowledge, and the appropriate reasoning mechanisms to both answer questions and assimilate new information. There are four criteria to evaluate a knowledge-representation scheme: flexibility, user-friendliness, expressiveness, and efficiency of processing. Flexibility, user-friendliness, and expressiveness are required to simplify the tasks of programming and comprehension. The efficiency or tractability of a knowledge-representation scheme dictates the efficiency of the solution to the application. Much of the research in this area represents a tradeoff between expressiveness and tractability.

Despite a great deal of effort devoted to research in knowledge representation, very little scientific theory is available to either guide the selection of an appropriate representation scheme for a given application or transform one representation into a more efficient one. Although a number of knowledge-representation schemes have been proposed, none is clearly superior to the others for all applications.

The following sections present two attributes for comparing knowledge representations: local versus distributed aspects, and declarative versus procedural. The classical knowledge-representation schemes are also evaluated on the basis of these features.

#### *1) Features of Knowledge Representations*

*a) Local versus distributed representations:* In a local representation, each conceptual datum is stored in a separate hardware unit. A word or item of data stored in a register is an example of local storage for that data item. As a result, the data are simple to read, update, and understand. Unfortunately, if any hardware unit fails, all knowledge contained in that unit is lost to the system. Most current systems, symbolic and numeric, utilize local representations for individual pieces of data.

In a distributed representation, a piece of knowledge is represented over many units, and each unit may be shared among multiple pieces of knowledge which correspond to features of multiple concepts or data items. The advantage of such a representation is that it is fault tolerant. If a small proportion of units fail, the integrity of the distributed data undergoes little change. This property is very attractive for practical implementations. Distributed representations also allow for a great deal of parallelism in computation [191]. However, they are usually harder for a user to understand and modify.

Table 2 summarizes the salient characteristics of local and distributed representations. It should be noted that there is not a concrete boundary between local and distributed features. Some features in a knowledge-representation

Table 2 Attributes of Local and Distributed Representations

| Attribute | Local | Distributed |
|---|---|---|
| Storage technique | Each data stored in dedicated hardware | Data represented over multiple units |
| Ease of understanding | Easy for humans to comprehend | Difficult for humans to interpret |
| Modification of stored data | Simple | More difficult |
| Fault tolerance | Loss of hardware results in loss of all stored data in this unit | Loss of small proportion of units does not seriously damage integrity of data |

scheme are local, while others are distributed. At one extreme is a standard implementation of a simple Lisp program, for instance, which can be thought of as a hierarchy of local representations. The program is stored as one unit of information. The data structures used by the program are also stored as a single entity. Finally, each piece of data within a data structure is stored in one memory location. On the other extreme is a standard implementation of a neural network. A predicate logic program, however, is neither fully local nor fully distributed. The complete program is not a single entity but a set of logic statements. Each statement, however, is an example of a local representation. This is considered further in Section II-A2.

*b) Declarative versus procedural representations:* The issue of distributed versus local representations concerns the methodology for representing information in the computer. In contrast, the issue of procedural versus declarative representations distinguishes between techniques for representing the processing knowledge and processing methodologies employed by the computer programs.

A program written in a declarative representation consists of a set of domain-specific facts or statements and a technique for inferring knowledge from these statements. It is, therefore, characterized as a set of statements of knowledge *about* the problem domain. Examples of declarative representations include pure predicate logic and production systems (to be discussed in Section II-A2).

In a procedural representation, program statements consist of steps to be taken in the solution of the problem, which are statements of knowledge about *how* to solve the task. Examples of procedural program representations include the C language and Lisp.

Declarative representations are user-oriented and emphasize correctness and user-friendliness of programs. They are referentially transparent: the meaning of the whole can be derived solely from the meaning of the parts, independent of its historical behavior. This may increase programmer productivity [245] and result in tremendous potential for parallelism [88], [128].

Unfortunately, programs written in declarative representations are often inefficient to evaluate due to nondeterminism, implicit control aspects, and inconsistent knowledge. It is hard to add domain-specific knowledge and meta-level knowledge to declarative programs. The difficulty with using declarative representations to solve symbolic problems lies in determining how to *use* the facts stored in the program's data structures, not in deciding how to store them.

Procedural programs are not as user friendly as declarative ones because the programmer must specify all control knowledge. In addition, the validity of a procedural statement often relies heavily on other procedural statements in the program, which complicates both the creation and modification of software. The loss in flexibility in a procedural programming environment is counteracted by the inherent gain in ease of representing control knowledge. Procedural schemes allow the specification and direct interaction of facts and heuristic information, thereby eliminating wasteful search. Meta-knowledge can also be easily included in procedures. Overall, procedural representations are as much concerned with the technique and efficiency of the computation as with the ease of representing the domain knowledge.

The salient features of declarative and procedural representations are summarized in Table 3. As with distributed versus local features of a representation, practical knowledge-representation schemes may have both procedural and declarative features.

*2) Classical Knowledge Representation Schemes:* In this section, the classical knowledge-representation schemes are described and evaluated with respect to the features of local versus distributed, and declarative versus procedural qualities. Those that have received the greatest attention include predicate logic, production systems, semantic networks, frames, procedural languages, and fully distributed representations.

*a) Predicate logic:* Predicate logic studies the relationship of implication between assumptions and conclusions. Logic often seems a natural way to express certain notions, and there are standard methods of determining the meaning of expression in logic formalism [125]. Logic is useful for exploring the epistemological problems that determine how the observed facts can be represented in the memory of a computer without being concerned with the use of the knowledge. The major disadvantage of logic stems from the separation of representation and processing.

*b) Production systems:* Production systems use collections of rules to solve problems. These rules consist of condition and action parts, or antecedent and consequent parts [173]. It has been found that production systems provide a useful mechanism for controlling the interaction between statements of declarative and procedural knowledge. For this reason, production systems have been used extensively in expert systems and knowledge engineering. Unfortunately, the expressive power of production systems is limited. Some researchers have argued that rule-based expert systems cannot achieve expert-level behavior [59]. Another problem with production systems is their inefficiency due to high control overhead.

*c) Semantic networks:* A semantic network is a directed graph whose nodes represent objects, concepts, or situations, and whose arcs represent relationships between nodes [183]. The basic inference mechanism in semantic networks is "spreading activation." The idea here has a clear neural inspiration: certain concepts in memory become a source of activation, and activation spreads in parallel to

**Table 3** Attributes of Declarative and Procedural Representations

| Attribute | Declarative | Procedural |
|---|---|---|
| Emphasis | Knowledge of domain | Knowledge of solution |
| Technique | Domain-specific statements | Solution techniques |
| Orientation | User friendliness; Ease of understanding | Efficiency of solutions; Ease of representing control knowledge |
| Parallelism | Natural, but countered by unnecessary search | Constrained and often user specified |
| Control | Transparent to the user | Specified by the user |

related concepts. The significance of this graphical representation is in allowing certain kinds of inference to be performed by simple graph-search techniques. Yet simple semantic networks can only express a collection of variable-free assertions. Several authors have shown that semantic networks can be extended so that they have the same expressive power as predicate logic [126]. Frequently, semantic networks are used as data structures for manipulation using other knowledge-representation schemes (such as Lisp, which is a procedural representation).

*d) Frame representation:* Frame representations employ a data structure for representing stereotypical situations [158]. The frame-description form is mainly an elaboration of the semantic-network one. Its emphasis is on the structure of types themselves (called frames) in terms of their attributes (called slots). A frame includes declarative and procedural information in predefined internal relations. Attached to each frame is various heuristic information, such as a procedure on how to use the frame. Although many issues about the possible implementations of frame-based systems are unresolved, the basic idea of frame-like structuring of knowledge appears promising and has appeared in various forms in many conventional languages.

*e) Procedural representations:* In a procedural representation, a knowledge base is viewed as a collection of modules expressed in a procedural language, such as Lisp or C. The procedural scheme is capable of representing heuristic knowledge and performing extended logical inferences, such as plausible reasoning. Due to the elimination of wasteful search, this representation scheme can be carried out efficiently. However, it is often limited by the available constructs. Conventional Fortran or Pascal programs, for example, have been found to be inadequate in supporting efficient symbolic processing.

*f) Connectionist representations:* A connectionist representation is a form of distributed representation: concepts are represented over a number of modules or units. When presented with input, units that have a positive correlation with an input feature activate, and those with negative correlation exhibit inhibitory signals. In this fashion, input can be recognized as a function of connection strengths among units (see also Section III-B2) [191]. Distributed representations allow automated procedures for learning concepts and representations and have great potential for parallelism in computation. Their major drawback lies in the difficulty of interpreting the system state and the internal representations. Additionally, the programming of these computers often requires a lengthy training period.

A given representation may exhibit local or distributed and declarative or procedural aspects at different levels of the representation. Table 4 summarizes the characteristics of these representations and categorizes the representations by the hierarchy of knowledge representation inherent in the technique.

## B. Knowledge Processing

Different reasoning methods are associated with different knowledge-representation schemes and require different architectural supports. Table 5 shows the classical knowledge-representation paradigms and their respective reasoning techniques.

Table 5 Reasoning Techniques

| Representation | Typical Reasoning Technique |
|---|---|
| Logic | Resolution (unification) |
| Production rules | Forward/backward chaining |
| Semantic networks | Spreading activation |
| Frames | Procedural attachments |
| Procedural | Control flow |
| Connectionist | Propagation of excitation |

It is argued that humans use logic-like reasoning in the domain of rational knowledge and apply memory-based reasoning for perceptual actions. For over 30 years, logic-like deduction has been the dominant paradigm in AI

Table 4 Examples of Knowledge-Representation Schemes

| Representation | Level of Representation | Characterization |
|---|---|---|
| Logic | Variable | Local/Declarative |
| | Statement/Relation | Local/Declarative |
| | Program | Distributed/Declarative |
| Production System | Variable | Local/Declarative |
| | Statement/Relation | Local/Either |
| | Program | Distributed/Declarative |
| Semantic Networks | Node | Local/Declarative |
| | arc/relation | Local/Declarative |
| | Network | Local/Procedural |
| | Program | Distributed/Declarative |
| Frames | Variable | Local/Declarative |
| | Statement | Local/Either |
| | Slots | Local/Either |
| | Frame | Local/Declarative |
| | Program | Distributed/Declarative |
| Procedural | Variable | Local/Either |
| | Statement | Local/Procedural |
| | Program | Local/Procedural |
| Connectionist | Connection Strength | Local/Distributed |
| | Propagation Technique | Local/Procedural |
| | Data and Knowledge | Distributed/Declarative |

research. This paradigm has been applied to a wide range of problems, especially expert systems. Although intelligent behavior often resembles logic-like reasoning with limited search, the intensive use of memory to recall specific episodes from the past (rather than rules) could be another foundation of machine reasoning [209]. Memory-based reasoning (or case-based reasoning) does not use rules, but attempts to solve the problem by direct reference to memory. The Connection Machine is an example of a machine designed for memory-based reasoning, although it can also be programmed to perform logic-like reasoning [103].

In the use of a knowledge representation, the knowledge processing technique must be tailored to cope with the application requirements. The greatest need, in symbolic processing applications, is the ability to deal with uncertain, incomplete, or conflicting information. Techniques for dealing with this problem are discussed in Section II-B1. In Section II-B2, methods for exploiting parallelism are discussed.

*1) Uncertain, Incomplete, and Inconsistent Knowledge Processing:* The techniques for dealing with these problems in knowledge processing are detailed in Fig. 4. The rest of the section is devoted to a brief discussion of the entries in the figure.

*a) Uncertain knowledge:* Conventional knowledge-representation techniques based on predicate calculus and related methods are not well suited for representing common-sense knowledge. Explicit and implicit quantifiers are fuzzy, and the standard inference methods mentioned earlier make no provision for dealing with uncertainty. Two types of uncertainty have been studied. One comes from noisy data and the fuzzy meaning of symbols; the other is associated with uncertain inference rules.

Methods and theories of capturing uncertainty have been examined in recent years. Probability and Bayesian statistics is the fundamental basis of most approaches to this problem. This takes many forms reflecting different issues. Approaches and techniques include fuzzy logic [250], [251], confidence factors [30], Dempster and Shafer's theory of plausible inference [199], odds [60], and endorsements [38].

Dealing with uncertain knowledge is most frequently handled by two principal components. The first is a translation system for representing the meaning of propositions and other semantic entities. The second is an inferential system for arriving at an answer to a question that relates to the information resident in a knowledge base. Application of Bayesian statistics to expert systems follows this approach [60]. A confidence factor (CF), or certainty factor, is used to decide among alternatives during a consultation session. A CF of a rule is a measurement of the association between premises and actions. A positive CF indicates that the evidence confirms hypothesis, while a negative CF negates the hypothesis.

Dempster and Shafer's theory of plausible inference provides a natural and powerful methodology for representing and combining evidence. Ignorance and uncertainty are directly represented in belief functions and remain throughout the combination process.

Endorsements are records of information that affect a hypothesis' certainty. They can be propagated over inferences, but in a manner that is sensitive to the context of the inference.

*b) Incomplete and inaccurate knowledge:* A key feature of symbolic computations is nondeterminism, which results from the fact that almost any intelligent activity is likely to be poorly understood. This implies that no systematic, direct algorithms for solving the problems are available. When a problem becomes well understood and can be solved by a deterministic algorithm, the solution of the problem is no longer considered as "intelligent" [206].

The starting point of conventional computations is a deterministic algorithm. Since most symbolic processing applications are knowledge intensive, such a deterministic algorithm may not exist. Efficient solution of the problem, therefore, requires continual refinement of the computation technique and may employ various knowledge-acquisition techniques. When knowledge of the application domain is incomplete or uncertain, heuristic solutions are utilized [133], [134], [180], [181], [205]. A heuristic is knowledge capable of suggesting plausible actions to follow, or implausible ones to avoid. It is desirable to use concise and

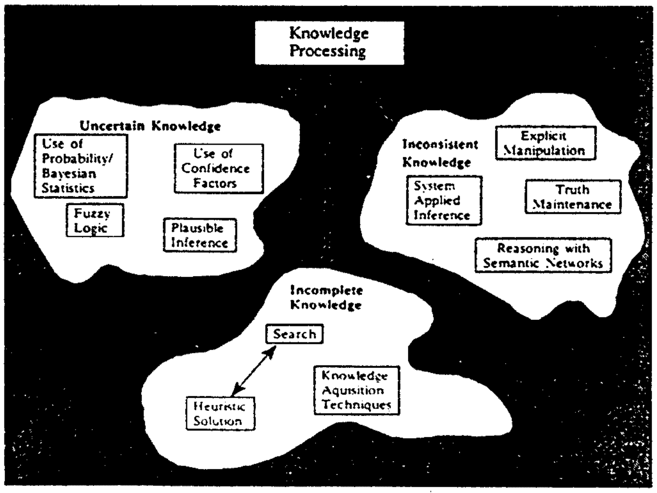

Fig. 4. Issues in knowledge processing and some applicable techniques.

accurate domain-specific knowledge and meta-knowledge. Unfortunately, this information is difficult to acquire in practice and, if available, may be fallible or tremendously large in size.

The nondeterministic nature of computations and fallibility of heuristic guiding may lead to anomalies of parallelism. As a result, when multiple processors are used, one or more of the processors may be guided by the heuristic-guidance function into a part of the search tree that is not explored in the same order as that in sequential processing. This out-of-order exploration of the search tree, coupled with the pruning of undesirable nodes, may result in a speedup (as compared to sequential processing) that is less than one or greater than the number of processors. Some results on when anomalies occur and how to cope with them can be found in the literature [129], [136], [141], [238].

In addition to heuristics, several new forms of logic for belief and knowledge have been introduced. Traditional reasoning methods suffer from the problem of *logic omniscience* [105]. Logic omniscience refers to the assumption that agents are sufficiently intelligent that they know all valid formulas. Thus, if an agent knows $p$, and that $p$ implies $q$, then the agent must know $q$. In real life, people are certainly not omniscient. The newly introduced forms of logic are more suitable than traditional logic for modeling beliefs of humans (or computers) with limited reasoning capabilities [63], [87].

*c) Inconsistent knowledge processing:* Traditional logic is *monotonic*. Monotonicity implies that new axioms may be added to the list of provable theorems only when they are consistent. Nonmonotonic reasoning provides a more flexible and complete logic system, as well as a closer model to human-thought processes. The motivations for nonmonotonic reasoning can be classified into two general areas: default reasoning, and reasoning in a changing environment [24].

Default reasoning can be broken into two distinct areas: exceptions to the rule, and autoepistemic logic. As the name implies, exception to the rule allows relations that contradict more general relations. A statement with *most* as a relation will not add any information to a monotonic logic system. Nonmonotonic systems allow a representation that includes exceptions to the general rule, without eliminating the validity of the computing environment.

Autoepistemic logic (a.k.a. circumscription and closed-world assumption) allows conclusions to be reached about relations for which no facts exist in the database. This involves the assumption that all relevant knowledge is in the database (closed-world assumption).

In a monotonic system, no modification of existing knowledge and data can be made without restarting all inference processes and results. In a world where new discoveries and revisions of previous beliefs is the norm, this is a poor model for a large knowledge/data-based system. Accommodating a changing environment is particularly important when default reasoning is used. A statement inferred by default may be corrected in light of additional evidence.

The distinguishing feature of different techniques for dealing with inconsistent knowledge is the method for handling correction of the knowledge base. These methods include explicit encoding, system applied inference, semantic networks, and truth maintenance. In explicit encoding, the programmer is responsible for writing code that will update the database when a new statement is added that may conflict with other statements [72]. In a system employing system applied inference, the system has user-encoded functions that automatically search for inconsistencies in the knowledge base [101]. McCarthy and Hayes have indicated how actions might be described using modal operators like "normally" and "consistent" [151]. Sandewall used a deductive representation of nonmonotonic rules based on a primitive called UNLESS [192]. Reasoning with semantic networks is another technique for ordering inferences and default reasoning, although they have been criticized for lacking a clear inference technique and for not being a sufficiently formal logic system. In Doyle's truth maintenance system (TMS), the reasons for program beliefs are recorded and maintained. These beliefs can be revised when discoveries contradict assumptions [58]. Improvements have been explored by de Kleer in his assumption-based truth maintenance systems (ATMS) [121]–[123]. In the IBM YES/MVS expert system, inconsistent deductions are automatically removed and new consequences are then computed in accordance with the changed facts [197].

*2) Parallel Knowledge Processing:* Humans are often thought of as the most efficient symbolic processing engines. Some researchers claim that symbolic problems can, therefore, be most effectively solved using techniques similar to those employed in the human brain. Observations of human intelligence suggest that human knowledge can be divided into perceptual and rational knowledge, each of which may involve different degrees of parallelism. In the perceptual stage of cognition, such as vision and speech understanding, massive parallel processing is possible due to the large number of independent data and simple control. Only limited parallelism can be exploited for rational knowledge. In other words, the degree of parallelism that could benefit the high-level reasoning is relatively small. Therefore, this kind of task should be solved by trying to accumulate heuristics, rather than trying to exploit parallelism.

Unfortunately, early experiences with symbolic multiprocessor architectures, such as Hearsay-II [70], Eurisko [135], and multiprocessor implementation of forward changing rule-based expert systems [75], have shown that parallel symbolic programs exhibit small speedups [119]. This has led to the possibly incorrect conclusion that symbolic programs written for sequential execution have low potential for parallelism.

The considerations of parallel knowledge processing are distinguished by four features: deterministic and nondeterministic parallelism, granularity of parallelism, data- and control-level parallelism, and user- and system-defined parallelism. These features are summarized in Table 6. Designs of parallel symbolic processors are presented in detail in Sections III and IV.

## III. Architectural Concepts for Symbolic Processing

With the symbolic processing application characterized and the representation technique for the solution of problems in that application selected, it is possible to choose the appropriate features and attributes for a computer system to solve problems in that application. An architectural component of a processing system is defined as a hardware or

Table 6 Issues on Parallel Processing

| Issue | Definition | Comments |
|---|---|---|
| Deterministic | Concurrent execution of multiple units of computation, all of which are necessary for job completion. | Low overhead guarantees speedup; tasks must be independent; pure functional programming is deterministic. |
| and Nondeterministic | Multiple potential solutions evaluated in parallel; parallelism used to replace or augment backtracking. | Easy to implement—always independent; nondeterministic nature may lead to anomalies in parallelism. |
| Granularity | Size of units of computation to be executed by a single functional unit. | Difficult to determine; a function of knowledge representation, problem complexity, the shape of the search graph, distribution of processing times, and the dynamic nature of the problem [140]. |
| Data Level | Data stored one element per processor, program executed in SIMD fashion. | Can be used for large database operations, sort, set operations, statistical analysis, . . . [240]. Can be implemented in memory—referred to as *active memory* [67]. |
| and Control Level | Independent control for parallel tasks. | Major type of parallelism used; MIMD systems; detection of parallelism can be more difficult than in numeric programs; nondeterministic nature requires dynamic mapping. |
| User Defined | Portions of program specified by users which can execute in parallel. | In numeric processing, DOACROSS is a typical example. The FUTURE construct in Multilisp is a symbolic construct [247]. |
| and System Defined | Parallelism detected and exploited automatically by the compiler or run-time software and hardware | Fully distributed representations allow massive system-level parallelism; some systems may employ both—the user only aids in indicating available parallelism. |

software structure that supports the solution of the application. In this section, current software and hardware architectures which are useful in symbolic processing systems are discussed. This section focuses on the specific architectures for symbolic processing and the way that they relate to fundamental design concepts.

Software architectures are comprised primarily of software languages and environments for encoding solutions to the application. Selection of a software environment imposes certain features that the software and hardware must support. The design process considers software and hardware implementations of the required features and selection of the best alternative, and is driven by a tradeoff between cost and expected performance improvement.

## A. *Software Architectures*

The area of software architectures for symbolic processing encompasses two important facets. The first is the design of appropriate software facilities, tools, and languages for the symbolic application, while the second facet concerns the tools used for mapping a symbolic application into software.

The discussion of software architectures is organized in the following manner. First, the process of designing software facilities and languages is analyzed. Following this is an overview of the most frequently researched and utilized programming paradigms for symbolic processing. Finally, the problem of mapping applications into software is overviewed.

• *The design of software languages:* The objective behind a software language is to provide software support and implementation of the knowledge representation(s) employed. As with the choice of the knowledge-representation scheme, the major goals are ease and ability to represent the solution of the application and the promotion of efficient execution of the algorithm. Once a technique, or techniques, for representing knowledge is selected, the major features of the language become apparent. For example, the choice of a logic representation dominates the characteristics of the Prolog language.

The software technique for implementing these features may not be as clear, however. For instance, Lisp was a procedural language developed for symbolic processing. In its design, functional programming with recursion and list-structured data were selected. These are not obvious choices, but they arise out of the use of a procedural representation. The selection of these features is also made on the basis of ease of representation and efficiency of processing. As another example, additional "impure" features may be added to the implementation of a knowledge-representation scheme for efficiently supporting computation and/or providing flexibility to the user. CUTs and side effects in Prolog are well-known examples. This aspect can be seen as a procedural addition to a logic representation, which is done to support the efficient implementation of logic programming.

The extension of conventional von Neumann computer languages for symbolic processing is an issue that has been explored extensively. By their nature, conventional computer languages are based on procedural representations. By examining the characteristics of symbolic processing, to be discussed in Section III-A1, the desirable features to incorporate into a conventional programming language become apparent. Such features include data structures, symbolic primitives, recursion, and others. Conventional languages which were designed for numeric processing, such as Fortran, have not proven to provide adequate support for symbolic processing. In particular, the languages are not sufficiently flexible to enable simple encoding of very complex symbolic operations. For this reason, the new and less conventional languages for symbolic processing are emphasized in this section.

As was observed in the preceding section, the emphasis in the design of new representations for symbolic processing problems has been focused in the area of adding declarative and distributed features to existing representation schemes. Part of the motivation for this emphasis is referential transparency, or freedom from side effects. This relieves some of the programming burden of the users, allowing easier programming of complex applications. In the following sections, three paradigms for the design of software languages which promote referential transparency are discussed: *functional languages* [16], [44], [97], *rule-based languages* [36], [52], [126], and *object-oriented languages* [81], [208], [210], [242].

*1) Functional Programming Languages:* The functional programming approach does not employ states, program counters, or other sequence-related computational constructs. A program is a function in the mathematical sense. The program, or function, is applied to the input, and the function is evaluated to the desired output. A functional approach can be thought of as a language based on Lambda Calculus; operators are applied to data or results of further function evaluations. John McCarthy's conception of list processing is viewed as a pioneering effort in this area [150]. Examples of functional language include pure Lisp [152], Backus' FP [16], Hope [32], Val [155], and Id [14].

In a functional language, the meaning of an expression is independent of the history of any computation performed prior to the evaluation of the expression (referential transparency). Precedence restrictions occur only as a result of function application. Notions such as side effects and shared memory do not exist in functional programs. The lack of side effects results in the determinacy property that is so valuable in parallel processing. Regardless of the order of computations of the arguments of a function, the same result (assuming termination) is guaranteed. Hence, all arguments and distinct elements in dynamically created structures in a functional program can be evaluated concurrently. For example, consider a simple program for computing the average of numbers in a list $s$:

$$\text{average}(s) = \text{div}(\text{sum}(s), \text{count}(s)).$$

If we attempt to evaluate average(1.(2.(3.nil))), the computation of sum(1.(2.(3.nil))) can clearly proceed independently of the computation of count(1.(2.(3.nil))). The key point is that parallelism in fundamental languages is implicit and supported by their underlying semantics. There is no need for special message-passing constructs, synchronization primitives, or constructs for specifying parallelism. It has been reported that implementation of functional languages on a parallel computer seems easier than on a sequential computer [46].

Programming in functional languages facilitates specification or prototyping, prior to development of efficient programs. With a satisfactory specification, it is possible to develop an efficient program through *program transformation*. The idea is that the program specification should be systematically refined to produce the program. Because functional languages are referentially transparent, they can be refined as familiar mathematical forms. Another advantage of functional programming is that it can represent high-order functions; a function can be passed as an argument. A comparison of functional programming with von Neumann programming is presented in Table 7.

Pure Lisp is a functional language. Many dialects of Lisp, however, are not purely functional. Operations on global variables, property lists, input/output, and other features incorporated in these dialects create side effects. This is seen as necessary to support efficient computation by avoiding recomputation of functions whose results are required in more than one place and to support convenient input/output. Unfortunately, the property of referential transparency is lost in most practical Lisp languages. Moreover, precedence restrictions are represented not only by functional calls, but also in procedures.

Table 7 Functional Versus Von Neumann Programming

| Functional Programming | von Neumann Programming |
|---|---|
| Programs are composed only of other programs. | Programs contain programs, expressions, and variables. |
| Programs can be freely built from others. | Programs are composed only with common data storage. |
| Same program can treat objects of different structure and size. | Changing size or structure of data means changing the program. |
| There is a strong theoretical background about programs. Programs may be proven as in mathematics. | Few general practical theorems exist about programs. Proving correctness of a program is extremely difficult. |

With the presence of side effects, it is not straightforward to identify the parallel tasks as in a pure functional language. Users are required to identify independent tasks with special primitives. Several parallel Lisp languages have been proposed and implemented. Multilisp, developed by R. Halstead at MIT and implemented on a 128-processor Butterfly parallel processor, includes the usual Lisp side-effect primitives for altering data structures and changing the values of variables [89]. Concurrency is introduced by means of the *pcall* and *future* constructs [41]. Both utilize an implicit fork-join. For example, (*pcall* A B C) will result in the concurrent evaluation of expressions A, B, and C; while (*future* X) immediately returns a pseudo location for the value of X and creates a task to concurrently evaluate X. The use of *future* allows concurrency between the computation of a value and the use of that value. The primitive *future* was introduced because the use of *pcall* alone did not provide a great deal of parallelism [90].

Proponents of functional languages believe that their simplicity and elegance will conduce to more orderly, more rigorous, more verifiable, and ultimately more efficient programming. Opponents worry about losing expressiveness as a result of the expression-evaluation-only model. The crucial disadvantage of functional programming is that it is difficult to represent the inherent nondeterminism in AI problems. The recursive formulation and leftmost outermost reduction of functional programs enable depth-first search naturally, but it is difficult to write a heuristic search program by a pure functional language, since heuristic search is inherently history-sensitive. In fact, heuristic search programs written in Lisp include many "*setq*" and "*do*" statements that are not pure functional primitives [246]. Due to the inability of representing nondeterminism and the inefficiency of dealing with large data structures, pure functional languages are often less suitable for general symbolic applications. Their usefulness for deterministic symbolic applications is, however, significant.

*2) Rule-Based Languages:* There are two major forms of rule-based languages available: logic and production systems. The languages associated with these representations are referred to as rule-based since both emphasize the relation between a condition and an inference or rule.

*a) Logic:* In its modest form, a logic program is the procedural interpretation of Horn clauses or predicate logic [125], [126]. Some ideas of logic programming, like automatic backtracking, have been used in the early AI languages QA3, PLANNER, and MICRO-PLANNER [18], [218]. The more contemporary language, Prolog, is based on logic programmer [37], [241]. Logic programming is a reasoning-oriented or deductive programming environment. Logic programming has received considerable attention because of its choice as the core computer language for the Fifth Generation Computer System Project in Japan [164].

The motivation of logic programming is to separate knowledge from control. However, logic programming implementations often include extralogical primitives to improve their run-time efficiency and flexibility in specification. For example, in Prolog, the CUT predicate is an extralogical control mechanism to define a similar construct as the *if-then* construct in conventional languages. In addition, variables in a logic program are often nondirectional, meaning that a variable does not have to be defined as an input or output variable at compile time, and its mode can be changed at run time depending on the context. As a result, dependencies among subgoals are not defined at compile time, and static detection of parallelism is very difficult. The solution is to require the users to specify the parallel processible tasks. In Parlog [35], every argument has a mode declaration that states whether the argument is input (?) or output (^). In Concurrent Prolog [200], a "read-only" annotation (?) is used. Users can also distinguish between "parallel AND" and "sequential AND" by using "," and "&", respectively.

Constructs can also be introduced to restrict parallelism until certain preconditions are satisfied. An example is a guard clause that has been adopted in Parlog. A guarded clause has the format: $h :- g | b.$, where $g$ is the *guard* of the clause and $b$ is its *body*. Subgoals in the body can only be evaluated when all subgoals in the guard have succeeded and values bound have been committed to the body.

User specification of parallelism certainly detracts from the objective of declarative programming. This is a problem even in the Restricted AND-Parallelism (RAP) model [51]. Although the user does not have to explicitly specify parallelism, the user must be aware of the underlying computational model. Both mode declarations in Parlog and read-only annotations in Concurrent Prolog impose a fixed execution order on subgoals, which may be inefficient. Choosing the proper subgoals in the guard is sometimes difficult and is not guided by any general principle. The distinction between "sequential AND" and "parallel AND," which is a linear order, is not sufficient to specify all precedence relationships, which form a partial order. Owing to the nondeterministic nature of AI applications, users cannot identify all parallel processible tasks perfectly. A better symbolic processing language should utilize both compile- and run-time detection of parallelism.

*b) Production system:* The other major form of rule-based language which promotes the separation of knowledge from control is based on production-system representation [28], [175]. A production-system program consists of a set of data and a set of rules that can act on the data. A rule is composed of a left-hand side (LHS) and a right-hand side (RHS). The LHS is the antecedent or situation and represents the conditions necessary for applying the rule. The conditions are in the form of a Boolean combination of clauses [28]. The RHS is called the consequent and indicates a set of changes to the data memory to be performed when the conditions of its LHS are met. Thus, a production system can be viewed as a combination of matching the logic condition and modifying the data in a procedural fashion. Strategies are required for matching data conditions with LHS of rules and for resolving conflicts when more than one rule has a match. The conflict set is the set of antecedents and their bindings that match elements in the working memory. Production systems operate in a recognize-act cycle. The recognize cycle computes the conflict set, while the act cycle selects one matching production and acts on it.

One of the popular programming environments for implementation of a production-system representation is the OPS5 system [28]. The OPS5 system highlights the issues involved in designing production systems. OPS5 employs data typing. The working memory (data) is viewed as a separate entity from the production memory, where the rules are stored. OPS5 employs a Rete match algorithm which computes the conflict set but does not select one production to act on. The algorithm works by storing the matching condition in the form of a tree. After the recognize-act cycle, rather than recomputing the entire conflict set, the tree is updated via tokens, which reflect the addition or deletion of elements from the working memory [28], [74]. OPS5 has two conflict-resolution strategies: LEX and MEA. LEX orders the conflict set on the basis of the recency of the time tags corresponding to the working memory elements that match the condition elements of the production rule. In contrast, MEA orders the conflict set using the recency of the working memory element that matches the first condition of the matching conditions (even if it is not a maximum).

Production systems provide a natural programming paradigm for "*if-then*" programming environments, such as those employed in expert systems. Unfortunately, algorithms with iterations and recursions are difficult to encode. In addition, rules are independent, and structural organization of programs requires special attention by users. Thus, it is difficult to develop large programs using production systems [186].

*c) On functional versus rule-based languages:* The advantages and disadvantages of functional languages stem from the procedural and formal mathematical nature of Lambda Calculus. In contrast, rule-based languages have complementary advantages and disadvantages which stem from their declarative nature.

The properties of nondirectionality of inputs and outputs, dynamic binding of variables, and nondeterminism make logic languages more expressive. On the other hand, with the aid of high-order functions, which permits quantification over individual data items as well as predicates and functions, functional programming enables more concise programs. Programs with high-order functions are easier to understand and reason about.

Although logic languages are more expensive, their implementation in a parallel environment is more difficult due to the nondirectionality of variables. This flexibility complicates the detection of parallelism at compile time and results in the dynamic behavior of execution of logic programs. Current Prolog systems also lack a means to describe the termination of computations on conceptually infinite data structures and the concept of "evaluation" of function invocations, which makes the logic base nontransparent. In contrast, the run-time behavior of pure functional programs is much simpler to control than that of first-order logic programs, particularly in a parallel context. Techniques, such as graph reduction and data flow, have been studied for parallel evaluation of pure functional languages.

Obviously, it would be advantageous if the simple control mechanism of functional languages could be applied to support languages with the great expressive power of logic languages. Considerable efforts have been devoted to combine functional and logic programming [52]. Some researchers are trying to simplify logic languages by introducing directionality of information in logic programs [187]. This approach will degrade the expressive power of logic programs to that of first-order functional programs.

The alternative approach is to extend functional languages so that they have the expressive power of logic languages while retaining the underlying functional simplicity. The addition of unification to the Hope language is one example [47]. Subrahmanyam *et al.* have proposed FUNLOG, a language that integrates functional and logic programming. FUNLOG provides the programmer with the flexibility of choosing between a backtrack-free computational framework and a logic computational framework. *Semantic unification* has been introduced to serve as a basis for achieving the integration of function and logic, and can be used to replace the conventional unification procedure in logic programming [216]. TABLOG, a new approach to logic programming designed by Malachi *et al.*, is based on first-order predicate logic with equality and combines rule-based and functional programming. The use of this richer and more flexible syntax overcomes some of the shortcomings of the Prolog syntax [146]. Other languages that combine features of Prolog and Lisp include LOGLISP, QLOG, POPLOG, Qute, and Lambda Prolog [187].

*3) Object-Oriented Languages:* New languages and programming systems are being developed to radically simplify AI programming. *Object-oriented programming* holds promise as a programming framework that can be extended to concurrent systems, databases, and knowledge bases.

In conventional software, data and procedures are the main focus of the representation and are treated as separate entities. The choice of procedures and data is made by the programmer. In an object-oriented system, there is only one entity: the *object*. Objects may be manipulated like data, or describe manipulation such as a procedure, or both. Processing is performed by sending and receiving messages to and from the object that possesses the appropriate information. A selector in the object specifies the kind of operation. Message sending is uniform, and a message represents only what the sender wants (or the result returned) but does not include information on how to accomplish that. Objects respond to messages using their own procedures (called *methods*) for performing operations. Since all communication is done via messages, one method may not "call" another method. The technique for representing the procedural knowledge can be any knowledge-representation scheme, although most implementations of object-oriented languages have employed procedural representations of control knowledge within the object.

In addition to objects and messages, object-oriented languages may also employ the concepts of class and instance. A class is a description of similar types of objects. Using classes, attributes of objects may be shared. In addition, classes provide a mechanism for inheritance or implicit sharing. Inheritance is used to define objects that are almost like other objects. Thus, classes provide an interface for the programmer to interact with the definition of objects.

*Data abstraction* is an important principle that is entailed through message sending. Object-oriented languages support both the management and collection of data through *abstract data types*, and the composition of abstract data types through an *inheritance* mechanism.

The requirement of typed data abstraction with inheritance is explicit and definitive, suggesting that object-oriented programming should be characterized by the nature of its type mechanisms rather than by the nature of its communication mechanisms. In a sense, object-oriented programming can be defined as

object-oriented = data abstraction + data types + type inheritance.

The object-oriented programming paradigm is a methodology mainly for organizing knowledge domains but is permissive in its methodology for communication. The message/object model provides no new leverage for expressing concurrent problems. Concurrent models, operating systems, and coordination tools can be built from such lower-level objects as process, queues, and semaphores.

Early exploration of object-oriented programming was found in Simula [42]. A more contemporary object-oriented language which has received a great deal of attention is Smalltalk [81]. A variety of object-oriented languages includes LOOPS [210], Actor [6], CommonObjects [208], OIL [48], and others [242]. Recently, CommonLoops has been suggested as a standard for object-oriented extensions to Common Lisp [23].

The *Actor* model, developed by Hewitt at MIT, is a formalization of the ideas of object-oriented language that also considers the added effect of parallelism [98]. An actor is the analogue of a class or type instance. Computations in the Actor model are partial orders of events, inherently parallel and having no assignment commands. The language Act3, based on the Actor model, combines the advantages of object-oriented programming with those of functional programming [5]. The Apiary network architecture has been proposed to support the Actor model [99], [100].

*4) Mapping Applications into Software:* Software development, an active area of research in software engineering, is the process of mapping the application into a language chosen for the given symbolic application. The process begins with the selection of a solution technique. This decision includes the choice of a knowledge representation and a method within that representation for solving the problem. Naturally, the available languages would greatly impact this choice. These choices may be referred to as requirement analysis in life-cycle models of software development [185], [201].

Software development environments can be classified into four generations [186]: discrete tools, toolboxes, life-cycle support and knowledge-based tools, and intelligent life-cycle support. Discrete tools were typical in the 1960s and 1970s and refer to the development of individual tool-like debuggers. Toolboxes refer to integrated packages of tools, the most prevalent example being Interlisp [224]. Life-cycle support and knowledge-based tools are being developed in the 1980s. Life-cycle support refers to software-development environments suitable for each stage in the design cycle, while knowledge-based tools try to incorporate domain knowledge to provide interactive assistance to the programmer. Finally, intelligent life-cycle support, a topic for future research [186], provides knowledge-based support for all stages of the software-development cycle in an integrated manner. Software engineering environments for distributed software development are also an increasingly important area of research [201].

With increased software complexity, verification, validation, and the enforcement of a structured discipline leading to reliable software are very important problems. Although a number of symbolic languages, such as Prolog, have been criticized for lack of structure, a programming style can be followed so that the resulting program is hierarchically developed, as in conventional structured programming languages, such as Pascal. There are mixed feelings on verifying and validating software written in symbolic processing languages. If these programs are treated as an algorithm and the requirements are well specified, then verification and validation are similar to those of programs written in traditional languages. Techniques such as test-case generation and path testing can be applied. However, it is difficult to test the validity of the knowledge used, since the knowledge may be heuristic and fuzzy in nature. The same criticism can be made about programs written in traditional languages. In this case, the validity of the program will largely depend on the experience of the experts and the procedures used in deriving the knowledge. Systematic knowledge-capture tools will help but will not guarantee proper collection and maintenance of consistent knowledge from multiple experts.

### *B. Hardware Architectural Support for Symbolic Processing*

The choice of the knowledge-representation schemes and the software languages largely dictates the desirable hardware architectures. In this section, the desirable intermediate hardware designs of a symbolic processing architecture are discussed. These hardware designs can support language-specific features, or primitive symbolic operations such as sorting and pattern matching. In Table 8, some of the features that require hardware or software support in Lisp and Prolog are shown.

Table 8 Features of Two Example Languages That Can Be Supported by Hardware

| Lisp | Prolog |
|---|---|
| Data typing | Condition matching |
| Function calls | Database functions |
| Recursion | Search |
| List structures—garbage collection | • search strategy |
| Individual commands—car, cdr, etc. | • backtrack mechanism |
| Parallelism—future, etc. | Unification |
| Application support | Parallelism |
| • application dependent | modes, guards, etc. |
| • database support | Application support |
| • transitive closure | |
| • and others | |

As with languages, hardware features can incorporate many well-established design philosophies, such as pipelining, parallel processing, microprogramming, and redundancy. Consider, for example, the hardware support of type checking in Lisp. In Fig. 5, the role that design concepts and

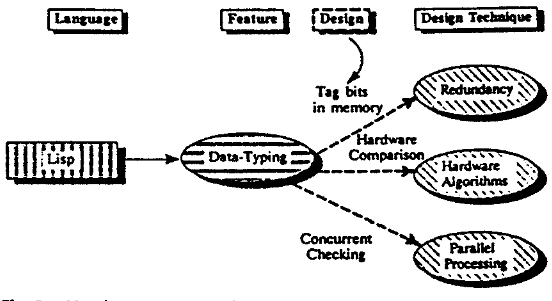

Fig. 5. Hardware support for data typing.

design requirements play in a hardware architecture is illustrated. The figure is intended to identify the role of concepts for approaching this design problem. It is not meant to say that hardware designers utilize such an approach in a real design.

Once a set of hardware and software alternatives has been enumerated, a subset of them must be selected for incorporation into the complete system. This will depend largely on the design philosophy of the system (see Section IV). The selection of competing structures (alternative designs that perform the same function) is made on the basis of anticipated gain in performance versus anticipated cost, both of which may be difficult to estimate, especially when the structures are not commercially available components. An approximate model is often used to guide the selection.

The current hardware architectures used to support symbolic computation can be classified into microlevel hardware features, subsystem-level hardware features, and system-level designs. A hardware architecture at the microlevel is a piece of hardware designed to support a feature of the language or processing techniques at its most fundamental level. It is specialized and does not, in general, provide useful computations outside of its role in the system. A subsystem-level architecture is an architecture that performs a complete and useful function by itself, but is often included as a portion of a larger system. A system-level design provides a complete hardware/software solution to a symbolic processing application. Microlevel and subsystem-level designs are discussed below, while system-level designs are presented in Section IV.

*1) Microlevel Hardware Features:* A microlevel hardware architecture is a unit dedicated to the support of a specific symbolic processing technique used in the system. As the number of features in symbolic applications and their languages is quite large, the number of corresponding possibilities for microlevel architectures is very large. Some microlevel architectures and the features that they support are presented in Table 9.

Five microlevel architectures are discussed in detail in this section: stacks, data tags, garbage collection, pattern matching, and unification. The remainder of the section is

Table 9 Functions That Can Be Supported by Microlevel Architectures

| Function | Example Architectures |
|---|---|
| Function calls, Recursion | Hardware stacks, Register windows, Fast memory techniques |
| Data typing | Memory tags, Concurrent tag-checking hardware |
| Sorting | VLSI sorter |
| Set intersection | Marker-passing systems such as NETL |
| Pattern matching | Systolic arrays, Content addressable memories, Finite state automata |
| Best-matching | Value-passing system such as THISTLE and neural networks |
| Garbage collection | Hardware pointers, Reference counters |

devoted to a discussion of emerging technologies employed in the construction of microlevel hardware.

*a) Hardware stacks and fast stack-access techniques:* Stack architectures support function calls. This is especially useful for Lisp and other functional programming languages. In the Symbolics 3600 computer [161], there are three stacks: the control stack, the binding stack, and the data stack, that are used to support tail recursion and shallow binding. To speed access to the control stack, the top several (up to four) virtual-memory pages of the stack are held in a dedicated fast-access 1K-word memory, referred to as the stack buffer. The stack buffer contains all of the current function environment (frame) plus as many of the older frames as fit. A second stack buffer contains an auxiliary stack for servicing page faults and interrupts without disturbing the primary buffer.

In ALPHA, a commercially available Lisp computer produced in Japan [94], a hardware stack is divided into four 2K-word physical blocks. Variables or arguments in a function of the Lisp program are stored in different locations of cell space or stack. A great deal of time is required to search for a free variable, especially in the case of deep binding. To speed the evaluation of the function, the hardware stack is designed to support value caching. The value for a variable is fetched from the envionment stack in the first access in a given function evaluation and stored in the value cache. Subsequent accesses will refer to values in the value cache. When the function exits, the variables it used in the value cache are marked invalid. Virtual stacks are also used to avoid the overhead of using a single stack and having to swap the entire stack in process switching.

Fast stack operations are also useful for implementing Prolog. Data are often pushed on and pulled from stacks in backtracking operations. However, the frames of a caller clause may be deeply buried in the stacks; hence, a stack architecture may not be adequate. In the Personal Sequential Interference Machine (PSI, a product of the Japanese Fifth Generation Compuer Systems Project), a cache memory is employed. Several operations suited to stack access are carried out in the cache memory.

*b) Tagged memory:* A conventional von Neumann computer does not distinguish between data and program. Both are stored as fixed-size binary words. Meaning is not inherent in the contents of storage but defined by the program manipulating the storage. A tagged architecture, however, relies on self-identifying representation at all levels of storage. Although tagging has been employed since the 1960s, early design considered tagging as a relatively unimportant and expensive peripheral concept. Tagged memory can be a key feature in symbolic processing computers today. Symbolic architectures often require identification of different types of physical and abstract data types, including integer, character, event, garbage, and others. During processing, it is necessary to identify the different operands employed in the computation. For this reason, the tagging of data to improve real-time type checking is appropriate.

The most common hardware support for data tagging is the allocation of extra bits in each word to represent its type. Data-type checking at run time may be supported by additional hardware and overlapped with regular processing. The speed of symbolic computers is often linked to how effectively they emulate a tagged-memory architecture [50]. Special hardware for data-type tagging has been estimated to increase system performance by as much as an order of magnitude in Lisp computers. Data tagging also supports garbage collection, facilitates better register utilization, reduces memory traffic, and simplifies the design of cooperating parallel processors and specialized functional units [71]. Data tagging is also essential in untyped languages since the programmer does not specify the type of instruction used. For instance, the programmer need not specify the type of an add instruction as integer, long, real, or double. The type of adder used at run time depends on the types of operands detected at run time, which are specified by the corresponding tags.

Data tags can be used to represent information other than just data-type. In the Classifier Machine [31], no addresses are used at all. Only tags are used to connect classifiers to each other. The no-addressing technique employed makes the Classifier Machine startingly different from classical von Neumann architectures.

*c) Garbage collection:* Garbage collection refers to the process of identifying memory cells whose contents are no longer useful for the computation in progress. In this case, memory cells are contiguous groups of at least one memory word. The process involves marking these cells as available for future use and compacting free memory into contiguous blocks. It has been estimated that 10 to 30 percent of the execution time in large LISP programs is spent on garbage collection. The implication on interactive or real-time systems is great, as garbage collection often requires large continuous segments of time from the CPU.

The initial techniques for automatic garbage collection centered around the use of reference counts [150]. Each cell had an extra field that indicated the number of times the cell had been referenced. The reference count was updated each time a pointer to the cell was created or destroyed. When the reference count reached zero, the cell could be reclaimed as garbage. These techniques have the advantage of intuitive simplicity and distributing the processing overhead along the processing of the task. However, the extra space and time required for use of the reference counts can be high, although some of the overhead can be shifted to compile time [19]. In addition, there is no way to reclaim cyclic structures. Generation scavenging is an important technique which reduces the overhead rate of garbage collection by using different rates for memory areas of different age (or generation) [233].

More recent research in the area of garbage collection has focused on parallel garbage-collection methods. Parallel garbage collection is garbage collection that is performed concurrently with program execution. Two processing entities are involved: the Mutator and the Collector. The Mutator is responsible for program execution, while the Collector is responsible for garbage collection. The techniques often center around the use of coloring cells [57], [127], or the division of memory space into (two) distinct regions [17], [142]. Tagging can be a useful microlevel hardware feature.

Parallel garbage-collection processors can be designed with very simple and fast components, without becoming a bottleneck of the system. In a design proposed by Hibino [102], the collector processor cycle time was 200 ns—six times faster than typical processor cycle times.

*d) Pattern matching hardware support:* Addition and multiplication are the mainstay of scientific computations. Similarly, pattern matching is the basic operation of symbolic processing. A pattern matcher may be employed for two major tasks (among other things): finding entries in a database and choosing an operation to execute next. For example, determining the applicable rule in a production system is a pattern matching problem. Empirical results show that 90 percent of execution time in a production system employed for expert systems can be spent in the matching phase [75]. Therefore, hardware pattern-matching support can simplify the programming task and improve runtime efficiency.

In most symbolic representations, symbols are represented in the form of strings. Conventional string-matching hardware can be classified into four categories. The first approach is associative memory. Although straightforward, it is difficult to operate on strings of variable lengths. The second technique is cascaded logic-memory arrays, also called a cellular array [137]. Each character in a pattern-string is stored in a cell and is compared with a character in the input string. A third technique is the finite-state-automata (FSA) method that uses a transition table to perform complex string matching. Finally, there is the dynamic programming technique that uses statistical characteristics of the general pattern in order to determine the parameter table for promixity matching. A survey of techniques for hardware support of pattern matching can be found in [165]. More recent techniques for hardware support have also been proposed [69], [220].

Pattern matching in some symbolic processors differs from conventional database retrievals because many symbolic applications contain widely varying field lengths and uncertainties in data may forbid exact matching of patterns. The pattern matching hardware for a symbolic processor must be tailored to the representation(s) for which it will be used.

In matching under uncertainties, best-matching is required. Best-matching structures search for the pattern which best matches the defined objective. Best-matching using associative memories has been explored [184]. Neural networks also provide potential for performing best-matching (see Section III-B2). Kanerva's Sparse Distributed Memory (SDM) is a system designed for best-matching [116]. The proposed prototype consists of a virtual memory that is addressed by a 1024-bit address, and a small physical memory. Each word in the physical memory has the 1024-bit address for this word and the data fields. When a memory address is given, all locations in the physical memory with addresses that differ by less than 450 bits of the given address are accessed, and the corresponding data fields are combined together into a single response. The 450 bits are chosen so that approximately 0.1 percent of the memory words will respond on the average for a physical memory of 4 Mwords.

In a semantic-network representation, pattern matching and other functions can be performed in parallel using a marker-passing operation [68]. A high-bandwidth communication channel is important for this type of pattern matching.

In a forward chaining rule-based production system, the objects to be matched are constants, and multipattern multiobject pattern matching is required. The Rete Match Algorithm is an efficient solution to this problem [74]. A number of hardware implementations of this technique have been proposed, including tree architectures [202], [212], SIMD Cellular Array Processor (CAP) [26], and tagged token data-driven multiprocessor [78]. It has been pointed out that the key architectural requirements to support the Rete Match Algorithm in parallel production systems are the use of a memory to maintain information across multiple recognize–act cycles, and the proper choice of granularity of parallelism [86].

*e) Unification hardware:* Unification, a form of pattern matching, is the fundamental technique in logic programming. It determines whether two terms can be made textually identical by finding a set of substitutions for variables in the terms, and replaces all occurrences of each variable by that variable's substitution. In general, both terms to be resolved in unification are allowed to contain variables; hence, unification can be thought of as a bidirectional pattern matching operation [126]. Since unification is applied extensively and is known to consume over 60 percent of the execution time in sequential logic execution, it is desirable that additional hardware or firmware support be available.

The primitive operations in unification are: 1) search for the called clause, 2) fetch of arguments of the caller and called predicates, and 3) examination of equivalence of arguments. In order to carry out unification in hardware fast memory access is required. To support dynamic memory allocation, an efficient garbage collection technique is also required. Finally, hardware support for data-type checking can also speed performance.

Research in this area has concentrated on string-matching hardware [204], uniprocessor machines [170], special unification chips [188], and pipelined unification [169]. To reduce the required memory space and improve performance, the use of structure sharing [29] and techniques for structure copying [156] have also been explored.

The Parallel Inference Engine (PIE) developed at the University of Tokyo employs special hardware, referred to a UNIRED, for unification and reduction. UNIRED may b characterized by the following features: tagged memory high-speed local memory that can be accessed in paralle parallel hardware stacks, and dedicated internal buses. Th unify processor fetches a goal from a memory module an candidate clauses from definition memory. The unify pro cessor then attempts to unify them, generates new goal and returns these goals to the memory module.

Parallel unification is also an area of great interest. Pa

emory
y with
given
lds are
its are
emory
emory

tching
using a
com-
pattern

em, the
rn mul-
h Algo-
number
ve been
], SIMD
en data-
that the
e Match
use of a
e recog-
ty of par-

f pattern
program-
made tex-
variables
variable
rms to be
variables;
ional pat-
is applied
ent of the
desirable
available.
search for
caller and
valence of
hardware,
mic mem-
chnique is
data-type

ing-match-
70], special
on [169]. To
ove perfor-
hniques for
d.
at the Uni-
ferred to as
ED may be
d memory,
in parallel,
l buses. The
module and
e unify pro-
s new goals,
le.
nterest. Par-

allel unification can be performed either by unifying each term pair in two atoms simultaneously, or by finding many possible unifications concurrenty [164]. Unfortunately, the unification problem is proven to be log-space-complete in the number of processors. This means that it is not possible to perform parallel unification in $O(\log^k n)$ time using a polynomial number of processors for any constant $k$, where $n$ is the total number of nodes and edges of the directed acyclic-graph representation of the clauses. It has also been shown, however, that near linear speedup can be achieved in parallel unification. Thus, unification algorithms are parallelizable from a practical perspective [237]. Array architectures for parallel unification have been proposed, such as the Cellular Array Processor (CAP) [26], and others [204]. A mesh connected array of unifiers has been proposed to exploit AND-parallelism in unification and may achieve superlinear speedup [203].

*f) VLSI and emerging technologies:* Very Large Scale Integrated circuit or VLSI technology has been a major factor in the cost reduction and increased functionality of symbolic processing systems. The high degree of space-time complexity in AI and symbolic computations has necessitated the use of both parallel processing and VLSI technology. The development of specialized microelectronic functional units is among the major objectives of the Japanese Fifth Generation Computing project [77], MCC [73], and DARPA's Strategic Computing projects [45].

VLSI technologies allow a single-chip computer to be realized. Although many functions can be implemented on a single chip, the size of the chip and the number of input/output pins are usually limited, and chip area has to be carefully allocated to achieve the highest performance. Reduced Instruction Set Computers (RISC) is a highly popular design approach that carries out only the most frequently used instructions in hardware and the less popular instructions in software [179]. The smaller chip area required by the control unit of RISC computers as compared to that of complex-instruction-set computers enables incorporation of these registers onto a single chip. It has been found that a large set of registers on the chip is a good design tradeoff to reduce the overhead of swapping registers in context changes.

SOAR, or Smalltalk on a RISC, is a project to develop a RISC chip for Smalltalk-80 [235]. The SOAR design details a 32-bit NMOS microprocessor containing 35 700 transistors and runs roughly 400 ns per instruction. Cycle time may be decreased to 290 ns if 3-$\mu$ lines were used. FAIM-1 is another project that designs RISC chips to carry out specialized functions in the system [12].

Besides implementing RISC on a single chip, specialized symbolic processing functions can be carried out in hardware as well. An example is the Texas Instruments Lisp chip with over 500K transistors on a 1-cm$^2$ chip implementing approximately 60 percent of the functions in a Texas Instruments Explorer [147].

Many existing computers for symbolic computation employ VLSI technology. The major building block of the Connection Machine CM-1 and CM-2 is a custom VLSI chip containing 16 processor cells [103]. The chip is implemented on a CMOS die about 1 cm$^2$ in area. There are approximately 50 000 active devices. Although each add takes approximately 21 $\mu$s, an aggregate maximum rate of 2500 MIPS or 5000 MFLOPS can be achieved with 64K processors implemented in 4K processor chips and 4K floating-point chips.

Cellular array structures are a widely studied technique that can take advantage of the available VLSI technology and exploit data-level parallelism in many symbolic processing problems. The Cellular Array Processor (CAP) is an example in this class [26]. A systolic cellular hardware design has been explored for performing unification [204].

There are also emerging technologies that will likely become cost effective in the near future for implementing computers for symbolic processing. These include GaAs circuits, wafer-scale integration (WSI), and optical computing techniques.

GaAs circuits have similar design requirements as conventional semiconductor circuits but are much faster. Switching speeds on the order of 10 ps have been reported in high electron mobility transistor (HEMT) GaAs circuits [130]. Gate propagation delays are typically on the order of 200 ps. Unfortuntely, fabrication of GaAs circuits is subject to limited size and greater numbers of defects, which adds to the already high cost [154].

WSI refers to the integration of multiple circuits on the same wafer in order to avoid the high performance and cost penalty of off-chip connections. As chip yield is low, the yield of a complete wafer would be negligible. Techniques such as focused ion beam (FIB) repair are utilized to increase yield of wafers. Low yields on WSI may make it more suitable for implementing a distributed knowledge-representation scheme, such as a neural network, in which the loss of a small fraction of the distributed knowledge may not be critical. When GaAs and WSI are combined, it is possible to implement a 32-bit GaAs processor on a single wafer [154]. Design of such a processor for production systems has been explored [131].

Optical processing can be used in a fashion similar to silicon gating. Switching speeds on the order of 5 to 10 ps are possible in optical gates. Optical circuits do not have the penalty of capacitance [244] and can communicate with low propagation delay and no interference. Optical processing has been developed in the form of arrays of light rays [21], [79] and optical crossbars [148]. Use of optics in storage media may greatly improve performance in symbolic systems with erratic memory behavior. It has been proposed that optical techniques have the potential for improving data- and knowledge-base processing speeds by two orders of magnitude [21].

In Table 10, the microlevel architectures presented in this section are summarized. Associated with each type of

**Table 10** Microlevel Architectures and Their Significance

| Architecture | Significance |
|---|---|
| Stacks | For function calls and value binding; more than one stack may be used. |
| Tagging | Data type checking; up to an order of magnitude speed improvement for Lisp. |
| Garbage Collection | Reclamation of usable storage; accounts for 10 to 30 percent run-time in Lisp. |
| Pattern Matching | Fundamental operation; up to 90% of execution time in production systems |
| Unification | Type of pattern matching for logic; over 60% of execution time in sequential logic programs. |

microlevel architecture is one of the main applications for that type of hardware, and an example of the significance of architectures for support of that feature.

*2) Subsystem Level Architectures:* The subsystem-level architecture represents an intermediate level between microlevel and complete-system designs. In this section, we identify three techniques for classifying the subsystem-level architectures that address different types of knowledge processing. Three different types of subsystems will emerge from this analysis: data- and knowledge-base machines, inference engines, and neural networks.

Control-flow, data-flow, and demand-flow are three important approaches that are used in the design of subsystem-level architectures as well as the complete system [236]. Their definitions and relative advantages and disadvantages are summarized in Table 11.

Knowledge representation plays an important role in the way that a complete system integrates its components. The Japanese FGCS project emphasizes logic representations and stresses the development of separate knowledge-base and inference engines, which are integrated after their development [9]. This viewpoint of the system architecture is illustrated in Fig. 6(a).

An alternative perspective is seen from the design of the Connection Machine [103], which closely reflects a semantic-network knowledge representation and memory-based reasoning. All the knowledge in the system is embodied in a large collection of facts; no intentional knowledge, or rules, are employed. For this type of system, the database is implemented directly on the architecture, and inferences are carried out in software and message exchanges. This perspective of system design is illustrated in Fig. 6(b).

The last design perspective is that of a hierarchical nature as shown in Fig. 6(c). An inference machine handles all meta-level inferences, while a knowledge-base computer deals with rules manipulating domain knowledge. A separate database computer carries out search and selection operations on the domain knowledge.

*a) Database architectures:* Early studies on specialized database architectures emphasized the use of parallelism. Such designs include CASSM [215], RAP [178], and DIRECT [56]. Some later systems, such as the Connection Machine, are designed with massive parallelism for symbolic applications and can be applied for a number of specialized database functions [240]. A comparison of several early parallel

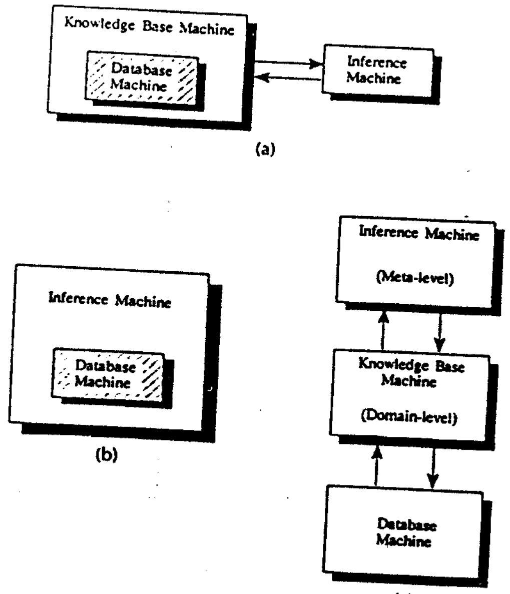

Fig. 6. Relationship among knowledge-base machine, inference machine, and database machine. (a) Japanese FGCS model. (b) Connection machine model. (c) Hierarchical model.

database computers and a survey of commercially available database computers can be found in [93] and [145]. Commercial manufacturers include Britton-Lee, Hitachi [227], International Computers Limited [15], and Teradata [171].

These systems may not function well for some applications because the bottleneck in database retrieval has found to be disk input/output and not processor cycles. As a result, very intelligent database processing using application-dependent knowledge and indexing may be preferable to massive parallelism [214].

*b) Knowledge-base architectures:* The objectives and requirements of a knowledge-base computer are different from those of a database architecture. The most prominent differences are noted in Table 12. An evolving knowledge-base subsystem should have a mechanism for either rejection of or truth maintenance for the insertion of incon-

**Table 11** Control-Driven Versus Data-Driven Versus Demand-Driven Computations

| | Control Flow (control-driven) | Data Flow (data-driven) | Reduction (demand-driven) |
|---|---|---|---|
| Definition | Conventional computation; statements are executed when a token of control indicates that they should be evaluated | Eager evaluations; statements are executed when all of their operands are available | Lazy evaluation; statements are only executed when their result is required for another computation |
| Advantages | Full control<br>Complex data and control structures are easily implemented | Very high potential parallelism<br>High throughput<br>Free from side effects | Only required instructions are executed<br>High degree of parallelism<br>Easy manipulation of data structures |
| Disadvantages | Less efficient.<br>Difficulty in programming<br>Difficulty in preventing run-time error | Time lost waiting for unneeded arguments<br>High control overhead<br>Difficulty in manipulating data structures | Does not support sharing of objects whose local state changes<br>Time to propagate demand tokens. |

Table 12 Differences Between Databases and Knowledge Bases

| Issue | Database | Knowledge Base |
|---|---|---|
| Contents | Collection of data and facts | Higher level of abstraction<br>Classes of objects |
| Complexity | Stored items are simple | Stored items are complex relations |
| Time dependence | Data changes over time | Knowledge changes less frequently with exception in situation knowledge |
| Size | Large number of facts | Fewer relations on classes of objects |
| Use | Operational purposes | Analysis, planning, ... |

sistent data or rules. Support for the inference mechanism is also desirable and may take the form of an automatic rule-selection mechanism, logic support, or special hardware for operations such as joins and projections of relations. Finally, the interface to the host computer should preferably be intelligent and may draw on the resources of the knowledge-base system.

The issues in the design of a knowledge-base computer include:

a) *Storage and manipulation of intentional and extensional data:* Extensional data refers to data representing facts, that is, statements with no quantified variables. Intentional data refers to general facts or rules. In Fig. 6(c), the extensional and intentional data are stored and operated on separately. In Figs. 6(a) and 6(b), they are processed by the same physical entity.
b) *Relational operations:* As above, hardware and software support must be provided.
c) *Hierarchical storage:* Knowledge can be classified into categories by degree of generality: fact through the most general meta-knowledge. The access characteristics are highly dependent on the type of knowledge accessed. A hierarchical storage for meta-knowledge may be used for efficiently exploiting the knowledge structure.
d) *Access-control algorithms:* A knowledge-base subsystem may be required to control access to its contents. The main issues here are security, integrity, and concurrency control.
e) *Parallel and Distributed processing:* Database updates are history-sensitive. It remains an open issue as to the best technique(s) for exploiting parallel and distributed processing in knowledge-base systems. Data-flow may be a good concept to apply.

In many applications, a database computer may be integrated with an existing host to form a knowledge-base computer. An example is shown in the Intelligent Information Resource Assistant developed at System Development Corporation [118]. The system consists of a Britton-Lee IDM 600 backend database computer, a Xerox 1100 workstation as a logic-based deductive engine, and a VAX 11/780 computer as a file and print server. This prototype has demonstrated that a knowledge-base computer can be easily constructed from existing hardware components.

The Japanese FGCS project has developed Delta, a combined knowledge and database computer. The motivation behind a specialized design is that the integrated system calls for a performance that cannot be met by commercial components. The system consists of a control processor, relational database engine, and a hierarchical memory. The control processor translates commands (received from an interface processor that communicates with a parallel inference machine) into subcommands that the relational database engine can perform. The relational database engine operates through a data-path with the hierarchical memory, which is composed of semiconductor and magnetic disk storage [166].

c) *Hardware support for inference engines:* Inference engines are a key component of knowledge processing architectures. Their structure is highly dependent on the knowledge representation and programming language employed.

An important problem in designing hardware support for inference engines is the architectural supports for searching the knowledge base. Deduction and search have been the dominant paradigms for machine inference over the last 30 years. As discussed in Section II, development of superior heuristics combined with efficient hardware is the best approach. The following are some of the key issues in research on search architectures. (More general issues on parallel processing have been discussed in Table 6.)

a) *Prediction of performance:* A major difficulty in developing search-based inference engines is the inability to estimate their performance without the actual execution of the search. This is due to both the nondeterministic nature of searches and anomalies in parallel search algorithms [238].
b) *Space-time tradeoff:* There is a space-time tradeoff in using heuristic knowledge. Very accurate heuristic functions may require greater amount of space and computation time than less accurate ones. This relationship, as well as the given architectural constraints, must be understood for the design of effective search subsystems.
c) *Architectural support for machine learning:* Heuristic functions used in search algorithms should be improved over time by automatic learning methods. Architectural support for nonmonotonic processing may be also helpful.
d) *Management of large memory space:* Heuristic search strategies may require large amounts of memory space to store the intermediate results and heuristic information. Tradeoff with respect to the effectiveness of a search strategy and the corresponding overhead incurred must be considered. Techniques for efficient memory management tuned to the search behavior are vital in such a system.
e) *Granularity of parallelism:* The proper choice of granularity is difficult to determine at design time due to the dynamic nature of the problem. Granularity may have to be varied at run time when more is known about the application.

f) *Scheduling and load balancing:* Due to nondeterminism in many search problems, direct mapping of a sequential search strategy into a parallel system may not result in the best performance. The key to effective scheduling is the proper order of execution, not just keeping the available processors busy. Counter to intuition, depth-first search is sometimes preferable to best-first search in real systems when memory constraints are considered [249]. Conditions for the sequential search strategy may also have to be relaxed to accommodate the architectural constraints [238]. For instance, the selection of the subproblem with the minimum heuristic value in a heuristic search may not be desirable when the overhead of selecting subproblems distributed in local memories of multiple processors is high. Selecting the local subproblem with the minimum heuristic value will suffice in most cases.

g) *Communication of pruning information:* When the search space is explored in parallel, excess computation may be performed if pruning information cannot be shared among the processors. In general, a tradeoff exists between search efficiency and communication overhead.

Microlevel hardware features discussed in Section III-B2 are often components of the inference subsystem. Dataflow, control-flow, and demand-flow techniques (see Table 11) have been employed in designing hardware supports for inference engines of Lisp and logic [10], [114], [167]. An example of a combined data-driven and demand-driven approach is demonstrated in ALICE [43] and Rediflow [117]. Although the theory behind data-driven and demand-driven computations appears very promising, no clear solution has been developed for many of the design considerations such as the proper granularity. It remains an open problem as to the proper tradeoff among demand-flow, data-flow, and control-flow in an inference computer.

*d) Artificial neural networks:* Automated computations based on a neural-network design philosophy originated many years ago. A pioneering contributor was Rosenblatt who developed the concept of perceptrons [189]. Recently, a great deal of interest has been revived in this area. Besides capturing the imagination by modeling computers after the human brain, artificial neural networks, or in short neural networks, offer a high potential for automated learning. The focus of storing information in the connections between neurons is why they may be referred to as connectionist systems.

A neural network can be viewed as containing eight components: processing units, state of activation, output function, pattern of connectivity, propagation rule, activation rule, learning rule, and an environment [190]. These are summarized in Table 13. The first design consideration is on the representation in the set of processing units. There are two possible approaches: local and distributed. A local representation allocates one concept to its own unit. Hopfield's network solving the traveling-salesman problem is an example of a local representation on a neural network [110]. While this may be an efficient solution to the problem, it does not perform learning. In contrast, in a distributed representation, each unit participates in the storage of many concepts [106]. In this way, a processing unit in a distributed representation stores an abstract feature at the microlevel [190].

Table 13 Artificial Neural Network Components

| Component | Description |
|---|---|
| Processing units | Three types; input, output, and hidden. |
| State of activation | Vector of the activation levels of the units in the system. |
| Output function | Function on the activation level of a unit which produces the units' output; may vary between units, but most systems are homogeneous. |
| Pattern of connectivity | The connections are what determines the performance and function of the system. |
| Propagation rule | A way of combining outputs of units and patterns of connectivity into an input for each unit; usually is a weighted sum of the inputs and the excitatory (+) and inhibitory (−) connection strengths. |
| Activation rule | A function for determining the new activation level of a unit on the basis of current activation and inputs to the unit. |
| Learning rule | Three types: develop new connections, abandon old connections, modify weights; only the last has been pursued; almost all learning rules are based on the Hebbian learning rule. |
| Environment | In which the computing engine functions. |

In a neural network, each unit has an associated activation level. This activation level can be analog, analog and bounded, or discrete. Its output is a function of its activation level. Output functions are usually threshold or sigmoidal functions. The output of each unit is distributed to a set of processing units. The emphasis that the outputs from other units have on a unit is determined by the weight of the connection between units. Finally, a node's activation is modified by the activation rule which is a function of inputs, their relative connection strengths, and the current activation level.

Classes of neural networks may be discerned by their learning paradigms [190]. One class is associative learning in which the network learns associations between the inputs and their desired outputs. The second class is regularity detectors, which learn to recognize interesting pattern on the input. An overview of designs for neural networks of both classes can be found in the references [4], [144]. Learning techniques in alternative neural network strategies are also extensively reviewed in the references [108].

Neural networks have been proposed for a variety of applications, although they are not yet widely used in practice. A neural network can be easily adapted as associative memory with capability for inexact matching. Its strengths include both speed and accuracy in the presence of noise [53]. At a more complex level, neural networks can be used for speech recognition [198], and vision tasks such as letter recognition [153]. Neural networks have been applied for solving combinatorial search problems, such as the traveling-salesman problem [110], [111], although solutions for large search problems on neural networks are not yet of as high quality as those of good digital algorithms [20]. Neural networks are also useful for some strategy-learning tasks. Experiments have been performed for a balance-control system [11].

The design of neural networks is still plagued by a number of difficult problems. First, a neural network must be trained for a given application and must be retrained when the system parameters change. There is no systematic method to generalize a neural network trained for one application and apply it for another application. Second, all the learning algorithms known today require extensive amount of training for good performance. For example, over 9000 learning sweeps, each of which has 40 phases, are required to train a shifter network using the Boltzmann Machine learning algorithm [107]. Moreover, the learning speed depends on the configuration of the neural network, which cannot be selected systematically. Dedicated hardware to emulate various configurations of neural networks and map the inner-loop operations into analog instead of digital circuits can improve the learning speed significantly. A million times improvement in speed has been demonstrated using hardware emulation [8]. Third, extremely large neural networks cannot be built with the current technology. There are numerous industrial efforts in building neural networks. For example, AT&T's R3 chip has a word size of 1048 bits and performs learning by back propagation. It was estimated that 4.5 million neurons can be built on one wafer by 2010 using 0.25 micron lithography and 250 million transistors [62]. Consequently, it is unlikely that a complete symbolic processor as intelligent as the human brain (with over 10 billion biological neurons) can be built with neural networks alone in the near future. Lastly, with the limited size of neural networks, it is necessary to partition the problem so that part of it can be learned by traditional methods and the other part by neural processing. However, knowledge representations in neural networks are drastically different from procedural and declarative representations in conventional symbolic processors. Systematic methods to integrate them are still missing.

## IV. Complete Systems

Up to this point, we have focused on individual techniques for effective symbolic computations. In this section, *complete systems* for solution of symbolic applications are classified into single-processor systems, parallel computers, and connectionist systems. As in the preceding sections, we are interested in analyzing the role of individual components in the design of complete systems.

There are two prevailing trends in designs. With the increasing complexity of many symbolic processing applications, there is a shifting emphasis on knowledge representations, as well as software languages and support, away from strongly procedural techniques and toward distributed and declarative computing environments. Parallel processing is another obvious trend in computer-system design for symbolic processing.

Depending on the starting point of the design, the system can be classified as following a top-down or a bottom-up approach. Top-down design begins with the specification and analysis of the application. A knowledge representation is then designed, tailored to the needs of the application. From this, a language is designed, and the system is mapped into software, microlevel and subsystem-level structures. This process may have to undergo many iterations if it is discovered that the functional requirements of the design cannot be implemented using the current technology. The FGCS project can be viewed as a system designed using a top-down design approach.

In contrast, a bottom-up design first selects the technology and design options such as data- or demand-driven calculation. A language and application suitable for implementation with these concepts are then sought. As above, the design process may have to undergo many iterations if it is found that the applications cannot be suitably supported by the computer designed. ZMOB [243] and the Butterfly Multiprocessor [27] are instances designed using this technique.

A shortcut to the top-down approach is to compromise between the top-down and bottom-up approaches, resulting in a middle-out design approach [239]. The middle-out approach begins with the selection of an appropriate and well established knowledge-representation scheme that is the most suitable for the application. The representation scheme should already have a well-developed programming environment that can be modified later for the needs of this specific system. Primitives for concurrent execution may be added to allow users to annotate concurrent tasks in the application. The hardware and software architectures are then designed, and the selection of features to incorporate includes consideration of previously used and designed structures. The middle-out approach can further be classified into top-first and bottom-first. In a top-first middle-out approach, the designers start with a well-defined knowledge-representation scheme and tailor it to the given application. ALICE and FAIM-1 are examples of architectures designed using this philosophy. In a bottom-first middle-out approach, the designers first develop the architecture to support a well-established representation scheme before mapping the application to the scheme chosen. DADO is an example developed using this approach.

The overall design process of a computer for symbolic processing is iterative (see Fig. 7). In mapping from application specifications to hardware and software technologies, the designers iterate in proposing and selecting knowledge-representation schemes, algorithms, and physical designs until a feasible and cost-effective mapping is found. Applicable design tools include methods for synthesis, analysis, optimization, heuristic designs, validation,

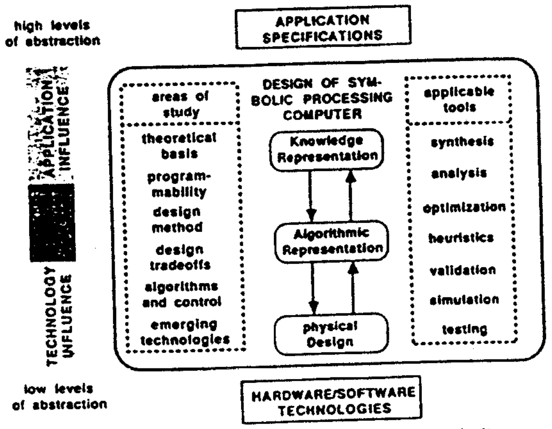

Fig. 7. Perspective of design of a computer for symbolic processing applications.

simulation, and testing. During the design process, the designers have to address issues related to theory, programmability, design methodology, design tradeoffs, effective control, and emerging technologies.

### *A. Single-Processor Symbolic Computers*

Lisp has enjoyed the longest tenure in the main stream of languages for symbolic processing and has led to the greatest number of computers devoted to its execution. The earliest Lisp computers were PDP-6s, followed by the PDP-10s and PDP-20s of Digital Equipment Corporation [152]. The half-word instructions and stack operations in these computers were particularly well suited to Lisp. A great deal of work has also been done specifically for garbage collection on the PDP-10s and 20s.

The MIT AI Laboratory introduced the design of a Lisp computer called CONS in 1976 [124]. This was soon followed by CADR in 1978, their second generation Lisp computer. This computer was the basis for many commercial Lisp computers, all introduced in 1981. These computers were the Symbolics LM2, the Xerox 1100 Interlisp computer, and the Lisp Machines Incorporated Series III CADR. Some notable recent commercial and research-oriented Lisp computers are shown in Table 14.

In addition to providing special hardware to improve efficiency, commercial Lisp computers also provide an integrated software development environment, such as KEE and ART, that allows programmers to develop, debug, and maintain large Lisp Programs. A recent trend is that these development environments can be implemented efficiently in software (rather than microcode) on high-speed general-purpose workstations. It is likely that in the future special-purpose Lisp computers will be used to execute rather than develop Lisp programs.

The design of special-purpose Lisp computers continues to be a popular research area. Many experimental computer designs have been reported [55], [83–85], [168], [182], [193–195], [222]. A single-chip design to support Scheme, a dialect of Lisp, has been demonstrated in the Scheme-79 and Scheme-81 chips [219]. Scheme-79 was limited by its implementation of a register file and slow programmable array logic. A redesign, Scheme-81, has proven to be much faster.

The popularity of rule-based systems has come at a time when parallel processing also became highly popular. As a result, most architectures designed for supporting these languages have been parallel computers. However, the study of single-processor rule-based systems is still important because a parallel system can be limited by the speed of its inferences, and the building block of a parallel system is likely to be a single-processor symbolic computer.

The three most notable designs for executing logic languages are SRI's pipelined Prolog processor [226], the Personal Sequential Inference (PSI) Machine of the Japanese FGCS Project [223], and the Cooperative High-Speed Inference Machine (CHI), also of the Japanese FGCS Project [230]. PSI, an integrated workstation with an execution speed of 30 KLIPS, is intended to be a software development tool for the project. PSI has been redesigned into PSI-II and has a performance of 150 KLIPS on the average [230]. CHI was designed with speed in mind, and took on a less con-

**Table 14** Notable Single-Processor Symbolic Computers

| Machine | Year/Status* | Primary Language | Features |
|---|---|---|---|
| Scheme-79, Scheme-81 | 1981/PO | Scheme | Single chip; tail recursion; lexical scoping. |
| Lisp Machines Lambda | 1983/CD | Zetalisp, LMLisp | NuBus—multiprocessor capability; stack orientation. |
| Symbolics 3600 | 1983/CA | Zetalisp, Flavors | Tagged memory; stack-buffer, hardware garbage collection; single-address instruction. |
| Tektronix 4400 | 1984/CA | Smalltalk, Franz Lisp | Lower-end AI workstation; Motorola 68010/20 |
| TI Explorer | 1984/CA | Common Lisp | NuBus; tagged memory; microprogrammed; mega-chip version has 60% of processor implemented in one chip with 550 000 transistors [147]. |
| Fujitsu ALPHA | 1983/CA | Utilisp | Value cache; hardware stack; virtual stack. |
| FGCS PSI-I, PSI-II | 1985/PO | KL0 (Logic) | PSI-II: CMOS-GaAs; TTL; 200 ns cycle time; cache; stacks; copying for structure data; hardware unification; tagged data; interpretive execution; 150 KLIPS average speed. |
| FGCS CHI-I, CHI-II | 1986/PO | Current Mode Logic (CML) | CHI-II: CMOS-GaAs; TTL; 170 ns cycle time; cache; about 400 KLIPS average speed for *append* operation. |
| SRIs Pipelined Prolog Processor | 1984/SI | Prolog | Pipelined execution; microprogrammed controller; interleaved memory; FCFS module queues. |
| SOAR | 1984/PO | Smalltalk-80 | RISC; hardware support for expensive procedure calls in Smalltalk-80; tagged and untagged instructions; large number of registers; automatic storage reclamation; direct object addressing; fast type checking. |

* Status Codes: (Tables 14–19) CA: Commercially available; PO: Prototype operational; UC: Under construction; PD: Paper design; CD: Discontinued commercial system; HS: Hardware simulated; SI: Simulations completed;

strained design technique. The original CHI was also redesigned into CHI-II, with an estimated improvement in performance from 280 to 400 KLIPS for the APPEND operation [230].

Single-processor support for production systems has focused on additional data memories [132] and RISC architectures [75].

Object-oriented languages have been implemented on sequential processors. The Xerox 1100 family of computers were one of the first workstations on which Smalltalk-80 was built. Smalltalk-80 has been implemented on a single chip using the RISC approach in the Smalltalk-On-A-RISC (SOAR) project [234]. There is no microcode or fine-grained addressing hardware, and few multicycle instructions.

### B. Parallel Symbolic Processors

In this section, we classify parallel symbolic processors in terms of their representations or programming techniques employed. Section IV-B1 discusses the methods of communication and synchronization in parallel symbolic processors. The rest of this section is devoted to discussions on parallel symbolic processors for functional, rule-based, and object-oriented representations.

#### 1) Communication and Synchronization

*a) Communication:* Message passing, marker passing, and value passing are three predominant communication methods in parallel symbolic processors.

Message passing is the conventional way of communication in which the information to be communicated is formulated into a message and sent over the interconnection network. The computing elements are generally complex, and the communication costs are high. Message passing is popular in many parallel symbolic processors.

Marker passing refers to the transfers of single-bit markers from one processor to another. A marker indicates the presence of a given property, and a set of markers indicate the conjuction of a set of properties. Each processor is simple and can store a few distinct marker bits. There is never any contention: if two markers arrive at the same destination, they are simply ORed together. The basic inference operation performed is, therefore, set intersection. Marker passing is especially suitable for implementing semantic networks and recognition problems in hardware. One such system is NETL [64], [66]. The Connection Machine was originally designed as a marker-passing system, but was modified later to carry out more powerful processing and communication features as well as support of virtual processors [68].

The last method of communicating information is value passing. In this form, information is passed as continuous quantities. Only simple operations are performed on these numbers. The salient feature of this approach is that if several values arrive simultaneously at one point, they are combined into a single output by a mathematical function. Consequently, there is no contention in information transfer. Examples of value passing systems include the Boltzmann machine [66] and other neural computation systems [109]. Iterative relaxation techniques for problems such as low-level vision, speech understanding, and optimization all seem to be suited to value-passing architectures.

*b) Synchronization:* Synchronization refers to the control of concurrent accesses to shared items in a parallel processing system. It is important in message-passing systems because messages may result in contention for shared resources. In contrast, synchronization is not critical in marker-passing or value-passing systems because there is a predefined method of passing markers and combining values so that contention will not occur.

Synchronization is important when there are shared data items. In a program written in a procedural language, the order of statements dictates the order of execution. If two statements share a variable, the first is to be executed prior to the second. Therefore, synchronization control is implicitly defined by the order of statements when data sharing is necessary. In contrast, the order of execution in a program written in a pure declarative language is not defined. When two tasks share a common variable, the order of execution is indeterminate. As a result, explicit specification of synchronization control is needed when data sharing is present. However, most declarative languages lack facilities for explicit specification of synchronization control.

Synchronization can be carried out by shared memory or message passing. Shared memory is popular and has been used in systems such as Aquarius [54], Concurrent Lisp Machine [217], Concert MultiLisp Multiprocessor [92], and the Parallel Inference Engine [82]. Blackboard architectures and shared variables are two applicable techniques for shared-memory synchronization.

The blackboard model was originally developed for abstracting features of the HEARSAY-II speech understanding system [61], [176]. There are three components: a set of knowledge sources, a blackboard, and control. The knowledge processing technique to solve the problem is partitioned into separate knowledge sources. The data, including input/output and partial solutions, are stored in the blackboard. The blackboard may be partitioned into smaller blackboards, forming a hierarchy of solution spaces. Knowledge sources manipulate the data in the blackboard in order to reach a solution. The only communication mechanism between these knowledge sources is the blackboard. A monitor is present to ensure that only one knowledge source is changing the blackboard at any time.

A more powerful blackboard architecture has been proposed in which control information (or meta-knowledge) is allocated a separate blackboard [95]. This approach is more flexible and suits the nondeterministic nature of symbolic processing.

Synchronization may also be achieved through shared-memory variables. Lisp languages that have been modified for parallel processing often contain shared variables for synchronization. Multilisp provides a mechanism for waiting for values to be generated in the future. As in other languages, procedure activations may not be well nested, and a process can terminate prior to an activation that it began. This is a problem that must be addressed by the programming system [91].

Single assignment languages, such as pure Prolog and pure dataflow languages, do not require careful synchronization of shared memory variables since a variable may be written only once [143]. In Prolog, the technique is to try to delay process reduction until information is available in order to make a better decision. In Guarded Horn Clauses (GHC) [231], the current kernel language of the Japanese FGCS project, OR-parallelism is not exploited. A strict synchronization rule suspends a subgoal if it tries to modify its parent environment. This simplifies the implementation of the language but results in a less expressive language [221].

Synchronization in a message passing system is accomplished through a protocol implemented in hardware or software. In a standard message passing environment, the messages may be of arbitrary complexity. This is more appropriate, for example, in systems using an object-oriented programming technique. Actor is a paradigm for systems with message passing of this nature [98], [100]. When an actor receives a message, it performs predefined primitive actions. In this sense, actor systems are inherently parallel. The Apiary architecture is based on actors [99]. Other message passing systems for symbolic applications include the Contract Net system [207] and the Rand Distributed Air Traffic Control System [33].

*2) Parallel Functional Programming Computers:* The majority of special-purpose parallel processors designed to support functional languages are oriented toward Lisp. Examples include Concert [92], EM-3 [248], and a multimicroprocessor Concurrent Lisp system developed at Kyoto University [217]. In all these systems, users are required to specify to some extent the tasks to be decomposed. Compilers for automatic detection of parallelism in sequential Lisp programs is an area of active research. Table 15 presents some of the more publicized parallel systems for functional programming techniques.

The majority of computers that were designed for general-purpose applications have only a few features specifically appropriate to symbolic processing. As a result, the inference engine and knowledge base are not separated and are almost exclusively implemented by sophisticated software structures. Lisp is added as one of the several languages to accompany their parallel computers. A Lisp compiler is used for decomposing tasks for parallel processing, and users are required to annotate tasks in various degrees. Examples of commercial multiprocessors supporting Lisp include the Butterfly [41], Connection Machine [39], and the Intel iPSC concurrent computer [22].

For the Connection Machine, special chips containing 16 bit-serial processors and router circuits were developed. *Lisp in CM-2 allows users to specify a parallel variable (*pvar*), which is a first-class object with value for each processor in the computer [40]. The primitive *pvar* can be accessed concurrently (with possible masks) by all local or remote processors in a SIMD or multiple SIMD mode. CM-Lisp is a dialect of Common Lisp extended to allow fine-grained, data-oriented parallel processing. It provides higher-level data abstractions called "zappings," which are similar in structure to arrays or hash tables. Broadcasts, reductions, and combinations can be specified.

Other general-purpose computers with limited support for parallel symbolic processing, such as ZMOB, are being developed at universities [243]. These computers can be viewed not just as symbolic processing computers, but also as general-purpose computers that are appropriate for both numeric and symbolic computations.

*3) Parallel Logic Architectures:* In this section, parallel systems suitable for evaluating logic programs are presented. A summary of notable projects is shown in Table 16.

Unification and search are two key features in evaluating logic programs. Architectures that emphasize efficient search of logic programs include the BAGOF architecture [34] and MANIP-2 [138]. The MANIP-2 architecture is particularly interesting due to its emphasis on heuristic parallel search strategies.

There are two significant parallel logic systems devel-

**Table 15** Notable Parallel Functional Programming Computers

| Machine | Year/Status* | Primary Language | Interconnection | Communication | Features |
|---|---|---|---|---|---|
| Butterfly | 1985/CA | Multilisp | Butterfly Switch | Shared memory | 256 MC68000-series PEs; homogeneous, tightly coupled; general-purpose multiprocessor. |
| iPSC | 1986/CA | Common Lisp | Hypercube | Message passing | 256 Intel 80286 and 80386 processors; no shared environment; user decomposes program into concurrent processes that communicate by messages; general-purpose multiprocessor. |
| Connection Machine | 1986/CA | *Lisp, CM-Lisp | Hypercube | Message passing | Model CM-2 has 4096 bit-serial processors; users annotate Lisp programs for SIMD or multiple SIMD parallel processing; C*, Fortran, and Paris (CM-2 assembly language) are also supported. |
| Concurrent Lisp Machine | 1983/PO | C-Lisp | Multiple buses | Shared memory | 17 MC68000-series PEs; special cell interface; control stack; garbage collector. |
| EM-3 | 1984/PO | EMLISP | Modified Delta network | Message passing | List-like data-driven language; 16 MC68000-series PEs; special router chip; control for function evaluation. |
| Concert | 1986/PO | Multilisp | Ringbus | Shared memory | 32-64 MC68000-series PEs; network is segmented bus in shape of ring. |
| Rediflow | 1984/SI | Functional Equation Language (FEL) | Mesh or richer connections | Message passing | Demand/data-driven; loosely coupled; hardware support for load balancing; distributed garbage collection. |
| Alice | -/UC | Hope, Lisp, Prolog | Cluster of processors, ring buffer. | Message passing | Transputer as basic processor; reference counter for garbage collection. |

*Refer to Table 14 for explanation of status codes.

Table 16 Notable Parallel Computers for Logic Representations

| Machine | Year/Status* | Interconnection | Communication | Features |
|---|---|---|---|---|
| BAGOF | 1984/PD | Bus | Shared memory | OR-parallelism; separate static and dynamic memory; token pool. |
| MANIP-2 | 1985/PD | Global broadcast bus | Message passing | Cluster of PEs with local memory; distributed selection; heuristic guiding and pruning. |
| Aquarius | -/UC | Bus and crossbar | Shared memory | Heterogeneous MIMD; 16 PEs; synchronization through Goodman Cache; crossbar to shared memory modules; special Prolog, floating-point, and I/O processors. |
| Parallel Inference Engine (PIE) | 1984/SI | Switching network | Shared memory | 100s to 1000s of inference units; goal rewriting model; OR-parallelism; sequential AND processing; Activity Controllers to control inference tree; Unify Processors connected to Definition Memory containing program. |
| Parrallel Inference Machine-Reduction (PIM-R) | 1986/HS | Multistage network | Shared memory | Many Inference Modules connected to Structure Memory units through network; structure copying. |
| Parallel Inference Machine-Dataflow (PIM-D) | 1986/HS | Multistage network | Shared memory | Multiple PEs connected to Structure Memory; unfolding interpreter; asynchronous communication; streams for nondeterministic control. |
| PIM-I | -/UC | Hierarchy | Shared memory, message passing | 100 PEs; 8-PE clusters interconnected with shared memory and parallel cache. |

*Refer to Table 14 for explanation of status codes.

oped at universities. The Aquarius multiprocessor developed at University of California, Berkeley, emphasizes a coupling of intensive numeric calculations and symbolic manipulations [54]. It intends to utilize parallelism at all levels of computation and considers cost as secondary to performance considerations. Another university project is the Parallel Inference Engine (PIE), being developed at the University of Tokyo [163]. The target is 1000 processors, and a speedup of 170 has been estimated for 256 processors. PIE utilizes OR-parallelism only.

Probably the most massive effort at the development of parallel logic systems is contained in the Japanese Fifth Generation Computing System project (FGCS). The project discerns three major areas of development: problem solving and inference machines (hardware), knowledge-base management systems (software and algorithms), and an intelligent man-machine interface [162]. The project is divided into three stages. The initial stage explores basic computer technology and processing techniques. This stage has been completed. The middle stage is for the development of subsystems, and the construction of experimental subsystems. The final stage is devoted to the development of the complete system.

The initial-stage designs of the Parallel Inference Machines (PIM) were based on two concepts: reduction and dataflow (PIM-R, and PIM-D, respectively) [115, 166]. The architectures for these computers were similar, but the technique for evaluation reflected these two philosophies. The hardware of PIM-R and PIM-D have both been simulated.

PIM-I is a hardware design for the intermediate stage of the FGCS project. The target speed for the 100-processor PIM-I is 10 to 20 MLIPS, with target speed of 200–500 KLIPS for the individual processors [230]. The machine language for this computer will be KL1-B, which is based on Guarded Horn Clauses. The software development will be done on a network of PSI systems (multi-PSI) [113].

*4) Parallel Systems for Production Systems Computations:* The exploration of computers for production systems has been carried out primarily at universities. Table 17 presents a summary of these projects.

The DADO1 and DADO2 projects [211], [213] at Columbia University develop a class of computers based on tree architectures. The upper-level nodes synchronize and select rules, intermediate nodes match and store rules, and the leaves are for the working memory.

Table 17 Notable Parallel Machines for Production Systems

| Machine | Year/Status* | Interconnection | Communication | Features |
|---|---|---|---|---|
| DADO1, DADO2, | 1986/PO | Binary tree | Message passing | DADO2; 1023 8-bit processors; 16K user memory; two modes; MIMD, and multiple SIMD; special I/O circuits. |
| Non-Von | 1985/PO | Binary tree with leaf connections; connections to Large PEs (LPEs) | Message passing | Binary SIMD tree of Small PEs; leaves are connected in mesh; LPEs connected by network, with connections to high-level nodes in tree; intelligent disk drives connected to LPEs. |
| PSM | 1986/SI | Shared buses | Shared memory | 32-64 processors; parallel Rete Match algorithm; PEs connected to memory modules through cache; local memory; hardware task scheduler. |

*Refer to Table 14 for explanation of status codes.

Table 18 Notable Parallel Computers for Object-Oriented Computation

| Machine | Year/Status* | Interconnection | Communication | Features |
|---|---|---|---|---|
| FAIM-1 | -/UC | Hexagonal mesh | Message passing | PEs = Hectagons; heterogeneous shared-memory multiprocessor: Instruction Stream memory, Post Office communication processor, Evaluation Processor, and others; three-port switch at edge of array for I/O and wrapping of connections. |
| Dragon | -/UC | Bus | Shared memory | Up to 10 32-bit workstation/processors; tightly coupled; associative cache at each processor. |
| Apiary | 1980/PD | Single-stage network | Message passing | Implements Actor model; computations in the Actor model are partial orders of events with no assignment commands. |

*Refer to Table 14 for explanation of status codes.

Another project at Columbia University is the Non-Von computer (Non-Von-1 is an earlier version) [104], [112]. Unlike DADO2, Non-Von connects smaller processing elements in a binary tree, which are subject to the control of large processing elements. Most of the pattern matching tasks that are done in the working memory have small granularity and are more suitable to be executed on a large number of small processing elements.

Finally, the PSM computer is a large-grain machine that is specifically designed to support the OPS5 system and a parallel Rete-Match algorithm. Simulations have shown promising speedups, and that 32 processors are sufficient to exploit most parallelism for this system [86].

Numerous studies on strategies of mapping productions systems to multiprocessors can be found in the references [132], [177], [225], [232].

*5) Parallel Object-Oriented Architectures:* Most development work on object-oriented programming has been on computers not specifically designed for object-oriented computation, such as the Intel iPSC. Two notable multiprocessors are designed specifically for object-oriented computations. FAIM-1 [12] is a multiprocessor with special RISC processors connected by a hexagonal mesh. OIL, an intermediate language, was first developed for interfacing modules. However, the scope of the project has recently been changed to exclude the development of OIL and base programming of the computer on MultiScheme. The Dragon project is another object-oriented design project, but supports only 10 processors [160]. Table 18 shows a summary of the notable projects.

### *C. Connectionist Processing*

The connectionist implementations focus on the correlation between nodes in a graph containing the knowledge and have been designed primarily for semantic networks. Other connectionist designs of interest are the artificial neural networks. Unfortunately, current technology for design of artificial neural networks have precluded their development in a role greater than that of subsystem.

The four designs of connectionist system for the implementation of semantic networks correspond to the three types of message passing environments. NETL utilizes the most elementary processing elements in a marker passing system [65], [66]. THISTLE is a similar design, but employs value passing instead [66]. The Connection Machine can be programmed to simulate marker passing in semantic networks and value passing in artificial neural networks using massive data-level parallelism [39]. Finally, SNAP relies on message passing [159]. These machine are summarized in Table 19.

### *D. Summary*

The purpose of this section has been to give a top-level perspective of existent special-purpose computers designed for symbolic processing. Sequential, parallel, and connectionist processing are three fundamental approaches to processing techniques that are appropriate for various knowledge representations. Thus, the first attribute of distinction for complete systems will be made on this basis.

Table 19 Parallel Connectionist Systems

| Machine | Year/Status* | Interconnection | Communication | Features |
|---|---|---|---|---|
| NETL | 1979/PD | Multi-level switching network | Marker passing | For semantic networks; million processors, each can store 16 markers; simple PEs; only Boolean functions. |
| THISTLE | 1983/PD | Multi-level switching network | Value passing | For semantic networks; similar to NETL, only with 8-bit value passing. |
| Connection Machine | 1986/CA | Hypercube | Message passing | General-purpose SIMD and multiple SIMD processing; can be programmed for marker passing operations in semantic networks and simulating artificial neural networks. |
| SNAP | 1985/PD | Mesh with global bus | Message passing | Square array of identical processors; CAM in each PE for relationships between nodes; Communication Unit; Processing Unit. |

*Refer to Table 14 for explanation of status codes.

The other primary discerning feature is the overall processing technique. Current systems define the processing technique in one of two ways. The first way is by knowledge representation, for example, computers for parallel logic computation. The second processing technique is based on the programming paradigm, such as functional or object-oriented paradigm. In either case, the processing technique can be used as the second attribute for distinguishing existent systems.

Systems can be further distinguished by various design decisions, and micro/macro-level architectures employed. For an overall perspective, however, it is not necessary to further classify these systems in order to understand the state of the art in symbolic processing systems.

Figure 8 classifies the computers presented in this section, on the basis of the above criteria. A number of systems

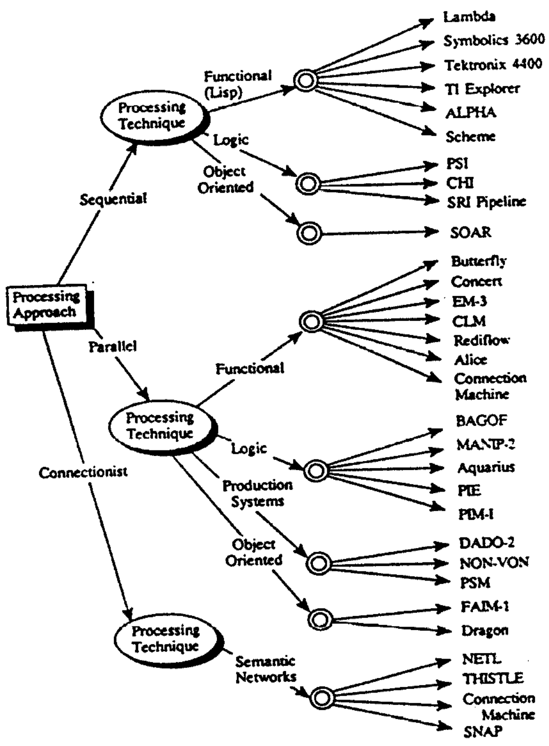

Fig. 8. Complete symbolic processing systems.

have been designed for diverse symbolic applications and cannot be uniquely classified into one category. The Connection Machine is one of the notable examples in this class.

## V. Research Directions

We have presented in this paper an extensive discussion and analysis of the state of the art in computer solution of symbolic processing problems. We conclude in this section by indicating some of the research areas where advancement will most likely benefit fast and efficient computer solution of these applications.

• *Technologies:* The basis for any computer system is the technology in which it is implemented. The design of a system is often driven by its cost; hence, the fastest technologies, subject to cost constraints, are used. New and emerging technologies may give higher performance but are often prohibitively expensive. The candidates that will likely become cost effective in the near future include GaAs circuits [130], [131], Wafer-Scale Integrated circuits (WSI) [154], analog-digital VLSI circuits, and optical computing techniques [21], [79], [244].

These emerging technologies offer tremendous potential for increasing the processing speeds of current computers. This extension in the limit of processing power is valuable, especially for real-time systems. However, the most that can be expected from these technologies is about one to two orders of magnitude speed improvement in the next ten years. They will not greatly impact the size or type of symbolic applications that are addressed today. Many of these applications involve huge search spaces of an exponential size; one to two orders of magnitude increase in computational speed will do little to extend the size of a solvable instance of such a problem [238].

• *Algorithms:* Research in the area of application-specific algorithms will have the greatest potential for speeding the solution of the given application. The development of new and improved algorithms for an application can be seen as finding alternative ways to incorporate knowledge about the application domain into the computer solution. In this way, advancement of symbolic processing capabilities in the area of application-specific algorithms is tightly linked to advancement in the area of knowledge representations.

• *Knowledge representations:* Most new knowledge representations for symbolic processing have emphasized declarative and distributed features in order to reduce programming complexity. These representation schemes may have to be modified or extended to tailor to the applications and computational environment. The addition of temporal features and nonmonotonicity would be helpful.

A major problem in the area of knowledge representations is the lack of an overall technique to guide the evaluation and selection of a knowledge-representation scheme. Research in this area could prove extremely valuable. Learning techniques for incorporating new knowledge about application domains into current solutions in a knowledge-intensive application may also have a great impact on symbolic processing. Artificial neural networks and connectionist representations are examples of incorporating automated learning techniques into the design philosophy from the knowledge-representation level.

• *Software architecture:* Software architectures are highl dependent on research in the area of knowledge repre sentations. Generation of new software environments tools, and languages will probably rely on amalgamation o known knowledge-representation techniques. Softwar development systems and automated intelligent program ming assistants represent prime areas for advancement o symbolic programming. The problems of program verifi cation and validation and continuous maintenance of sym bolic programs are important related topics.

• *Hardware architectures:* As with software, hardwar architectures are often based on known design technique such as parallel processing and pipelining. Innovation fo new architectural concepts may be caused by the avail ability of new and emerging technologies.

New hardware architectures are best utilized for ope ations that the computer performs frequently. Counter to intuition, identification of these tasks is very difficult. Operations may be instructions, parts of instructions, groups of instructions, or frequently recurring tasks. Identification of new and valid areas for development of new hardware architectures is an important area of research.

• *System design:* System-level design is often based on an overall design philosophy. Systems may contain, for example, a mix of data- and control-flow computation. The proper mix of control, data, and demand flow is one area of research that may impact systems for symbolic processing. New systems for symbolic processing may also greatly benefit from integrating new hardware subsystems and microlevel architectures, as well as the integration of new and emerging technologies. The major difficulty, however, lies in integrating designs with radically different knowledge representations. Combination of distributed representation offered by artificial neural networks and procedural or declarative representation offered by standard computers is an interesting area for development.

## References

[1] "ESPRIT: Europe challenges U.S. and Japanese competitors," *Future Generation Computer Systems*, vol. 1, no. 1, pp. 61–69, 1984.

[2] "Special issue on natural language processing," *Proc. IEEE*, July 1986.

[3] "Special issue on computer vision," *Proc. IEEE*, 1987.

[4] Defense Advanced Research Project Agency, *DARPA Neural Network Study*, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, MA, July 1988.

[5] G. Agha and C. Hewitt, "Concurrent programming using actors: exploiting large-scale parallelism," *Lecture Notes in Computer Science*, no. 206, pp. 19–41, Dec. 1985.

[6] G. Agha, *Actor: A model of concurrent computation in distributed systems*. Cambridge, MA: MIT Press, 1986.

[7] L. Aiello, C. Cecchi, and D. Sartini, "Representation and use of metaknowledge," *Proc. IEEE*, pp. 1304–1321, Oct. 1986.

[8] J. Alspector and R. B. Allen, "A neuromorphic VLSI learning system," in *Advanced Research in VLSI: Proc. 1987 Stanford Conference*, P. Loseleben, ed. Cambridge, MA: MIT Press, 1987.

[9] M. Amamiya *et al.*, "New architecture for knowledge based mechanisms," in *Proc. Int. Conf. Fifth Generation Computer Systems*, pp. 179–188, Japan, 1981.

[10] M. Amamiya *et al.*, "Implementation and evaluation of list-processing-oriented data flow machine," in *Proc. 13th Int. Symp. Computer Architecture*, pp. 10–19, 1986.

[11] C. W. Anderson, "Strategy learning with multilayer connectionist representations," *Proc. Fourth Int. Workshop Machine Learning*, pp. 103–114, June 1987.

[12] J. M. Anderson *et al.*, "The architecture of FAIM-1," *Computer*, vol. 20, no. 1, pp. 55–65, Jan. 1987.

[13] H. L. Andrews, "Speech processing," *Computer*, vol. 17, no. 10, pp. 315–324, Oct. 1984.

[14] Arvind, K. Gostelow, and W. Plouffe, "An asynchronous programming language and computing machine," Tech. Rep. 114a, Univ. of California, Irvine, CA, Dec. 1978.

[15] E. Babb, "Implementing a relational database by means of specialized hardware," *Trans. Database Systems*, vol. 4, no. 1, pp. 1–29, March 1979.

[16] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and algebra of programs," *Comm. ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[17] H. G. Baker, Jr., "Optimizing allocation and garbage collection of spaces," in *Artificial Intelligence: An MIT Perspective*, P. H. Winston and R. H. Brown, ed. vol. 1, pp. 391–396, Cambridge, MA: MIT Press, 1979.

[18] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, vol. 1, 2, and 3, Los Altos, CA: William Kaufmann, 1981, 1982.

[19] J. M. Barth, "Shifting garbage collection overhead to compile time," *Comm. ACM*, vol. 20, no. 7, pp. 513–518, July 1977.

[20] E. B. Baum, "Towards practical 'neural' computation for combinatorial optimization problems," in *AIP Conf. Proc. Neural Networks Computing*, pp. 53–58, 1986.

[21] P. B. Berra and N. B. Troullinos, "Optical techniques and data/knowledge base machines," *Computer*, vol. 20, no. 10, pp. 59–70, Oct. 1987.

[22] D. Billstrom, J. Brandenburg, and J. Teeter, "CCLISP on the iPSC concurrent computer," in *Proc. Sixth Intl. Conf. Artificial Intelligence*, pp. 7–12, 1987.

[23] D. G. Bobrow, *et al.*, "CommonLoops: Merging common Lisp and object-oriented programming," Tech. Rep. ISL-85-8, Xerox Palo Alto Research Center, Aug. 1985.

[24] D. G. Bobrow and P. J. Hayes, (ed.), "Special issue on non-monotonic logic," *Artificial Intelligence*, vol. 13, no. 1 & 2, April 1980.

[25] K. Bowen, "Meta-level programming and knowledge representation," *New Generation Computing*, vol. 3, no. 4, pp. 359–383, 1985.

[26] R. Brooks and R. Lum, "Yes, an SIMD machine can be used for AI," in *Proc. Int. Joint Conf. Artificial Intelligence*, pp. 73–79, 1985.

[27] C. M. Brown, C. S. Ellis, J. A. Feldman, T. J. LeBlank, and G. L. Peterson, "Research with the Butterfly multicomputer," *Rochester Research Review*, pp. 3–23, 1984.

[28] L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5*. Reading, MA: Addison Wesley, 1985.

[29] M. Bruynooghe, "The memory management of PROLOG implementations." in *Logic Programming*, K. Clark and S. A. Tarnlund, ed. pp. 83–89, New York: Academic Press, 1982.

[30] B. G. Buchanan and E. H. Shortliffe, *Rule-Based Experts Programs: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley, 1984.

[31] A. W. Burks, "Keynote of CONPAR86," *Lecture Notes in Computer Science No. 237*, pp. 1–17. New York: Springer-Verlag, 1986.

[32] R. M. Burstall, D. B. MacQueen, and D. T. Sannella, "HOPE: An experimental applicative language," in *Conf. Record Lisp Conf.*, pp. 136–143, 1980.

[33] S. Cammarata *et al.*, "Strategies of cooperation in distributed problem solving," in *Proc. Int. Joint Conf. Artificial Intelligence*, pp. 767–770, 1983.

[34] A. Ciepielewski and S. Haridi, "Execution of Bagof on the OR-parallel token machine," in *Proc. Int. Conf. Fifth Generation Computer Systems*, pp. 551–560, 1984.

[35] K. Clark and S. Gregory, "Note on system programming in PARLOG," in *Proc. Int. Conf. Fifth Generation Computer System*, pp. 299–306, 1984.

[36] K. L. Clark and S-A. Tarnlund, (ed.). *Logic Programming*, New York: Academic Press. 1982.

[37] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, New York: Springer-Verlag, 1981.

[38] P. R. Cohen and M. R. Grinberg, "A theory of heuristic reasoning about uncertainty," *The AI Magazine*, pp. 17–24, 1983.

[39] Thinking Machines Corporation, "Connection Machine Model CM-2 Technical Summary," Tech. Rep. HA87-4, Cambridge, MA, April 1987.

[40] Thinking Machines Corporation, "*Lisp Reference Manual*, Version 4.0. Cambridge, MA, Oct. 1987.

[41] A. S. Cromarty, "What are current expert system tools missing?" *Proc. IEEE COMPCON Spring*, pp. 411–418, 1985.

[42] O.-J. Dahl and K. Nygaard, "SIMULA—an ALGOL-based simulation language," *Comm. ACM*, vol. 9, no. 9, pp. 671–678, Sept. 1966.

[43] J. Darlington and M. Reeve, "ALICE: A multi-processor reduction machine for the parallel evaluation of applicative languages," in *Proc. Conf. Functional Programming Languages Comp. Architecture*, pp. 65–74, 1981.

[44] J. Darlington, P. Henderson, and D. Turner, *Functional Programming and Its Applications*, Cambridge, UK: Cambridge University Press, 1982.

[45] J. Darlington and M. Reeve, *ALICE and the Parallel Evaluation of Logic Programs*, Preliminary Draft, Dept. of Computing, Imperial College of Science and Technology, London, England, June 1983.

[46] J. Darlington, "Functional programming (Chapter 5)," in *Distributed Computing*, F. B. Chambers, D. A. Duce, and G. P. Jones, ed. London: Academic Press, 1984.

[47] J. Darlington, A. J. Field, and H. Pull, *The Unification of Functional and Logic Languages*, Tech. Rep. Imperial College, London, England, Feb. 1985.

[48] A. L. Davis and S. V. Robison, "The FAIM-1 symbolic multiprocessing system," in *Proc. IEEE COMPCON Spring*, pp. 370-375, 1985.

[49] R. Davis and B. Buchanan, "Metal-level knowledge: overview and applications," in *Proc. 5th Int. Joint Conf. Artificial Intelligence*, pp. 920-928, 1977.

[50] M. F. Deering, "Architectures for AI," *Byte*, pp. 193-206, April 1985.

[51] D. DeGroot, "Restricted AND-Parallelism," in *Proc. Int. Conf. Fifth Generation Computers*, pp. 471-478, Nov. 1984.

[52] D. DeGroot and G. Lindstrom (eds.), *Logic Programming*, Englewood Cliffs, NJ: Prentice-Hall, 1985.

[53] J. S. Decker, "Neural network models of learning and adaptation," *Physica*, pp. 216-232, 1986.

[54] A. M. Despain and Y. N. Patt, "Aquarius—A high performance computing system for symbolic/numeric applications," in *Proc. IEEE COMPCON Spring*, pp. 376-382, Feb. 1985.

[55] P. Deutsch, "Experience with a microprogrammed Interlisp systems," *Proc. MICRO*, vol. 11, Nov. 1978.

[56] D. J. DeWitt, "DIRECT—A multiprocessor organization for supporting relational database management systems," *IEEE Trans. Computers*, pp. 395-406, June 1979.

[57] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: An exercise in cooperation," *Comm. ACM*, vol. 21, no. 11, pp. 966-975, Nov. 1978.

[58] J. Doyle, "A truth maintenance system," *Artificial Intelligence*, vol. 12, no. 3, pp. 231-272, 1979.

[59] H. Dreyfus and S. Dreyfus, "Why expert systems do not exhibit expertise," *IEEE Expert*, vol. 1, no. 2, Summer 1986.

[60] R. O. Duda, P. E. Hart, and N. J. Nilsson, "Subjective Bayesian methods for rule-based inference systems," in *Proc. National Computer Conf.*, pp. 1075-1082, 1976.

[61] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty," *Computing Surveys*, vol. 12, no. 2, pp. 213-253, June 1980.

[62] F. Fagin, *How Far Can We Go With Proven Technology?* Special Session on Neural Networks, presented at AAAI Conference, Seattle, WA, July 1987.

[63] R. Fagin and J. Halpern, "Belief, awareness, and limited reasoning: preliminary report," in *Proc. Int. Joint Conf. Artificial Intelligence*, pp. 491-501, 1985.

[64] S. E. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge*, Series on Artificial Intelligence, Cambridge, MA: MIT Press, 1979.

[65] S. E. Fahlman, "Design sketch for a million-element NETL machine," in *Proc. 1st Annual National Conf. Artificial Intelligence*, pp. 249-252, Aug. 1980.

[66] S. E. Fahlman and G. E. Hinton, "Massively parallel architectures for AI: NETL, THISTLE, and BOLTZMANN machines," in *Proc. National Conf. Artificial Intelligence*, pp. 109-113, 1983.

[67] S. E. Fahlman, "Parallel processing in artificial intelligence," *Parallel Computing*, vol. 2, pp. 283-286, 1985.

[68] S. E. Fahlman and G. E. Hinton, "Connectionist architecture for artificial intelligence," *Computer*, vol. 20, no. 1, pp. 100-109, Jan. 1987.

[69] C. Faloutos, "Access Method for Text," *Computing Surveys*, March 1985.

[70] R. D. Fennell and V. R. Lesser, "Parallelism in artificial intelligence problem solving: A case study of Hearsay-II," *IEEE Trans. Computers*, vol. C-26, no. 2, pp. 98-111, Feb. 1977.

[71] E. A. Feustel, "On the advantages of tagged architecture," *IEEE Trans. Computers*, vol. C-22, no. 7, pp. 644-656, 1973.

[72] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3 & 4, pp. 189-208, 1971.

[73] M. A. Fischetti, "A review of progress at MCC," *IEEE Spectrum*, March 1986.

[74] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17-37, Sept. 1982.

[75] C. L. Forgy, A. Gupta, A. Newell, and R. Wedig, "Initial Assessment of Architectures for Production Systems," in *Proc. National Conf. Artificial Intelligence*, pp. 116-120, Aug. 1984.

[76] K. S. Fu, *Syntactic Methods in Pattern Recognition*, New York: Academic Press, 1974.

[77] K. Fuchi, "The direction the FGCS project will take," *New Generation Computing*, vol. 1, no. 1 pp. 3-9, 1983.

[78] J.-L. Gaudiot, S. Lee, and A. Sohn, *Data-Driven Multiprocessor Implementation of the Rete Match Algorithm*, Tech. Rep., Dept. of Computer Science, University of Southern California, Los Angeles, CA, 1987.

[79] T. K. Gaylord and E. I. Verriest, "Matrix triangularization using arrays of integrated optical Givens rotation devices," *Computer*, vol. 20, no. 12, pp. 59-67, Dec. 1987.

[80] M. R. Genesereth, "An overview of meta-level architecture," in *Proc. National Conf. Artificial Intelligence*, pp. 119-124, 1983.

[81] A. J. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Reading, MA: Addison-Wesley, 1983.

[82] A. Goto, H. Tanaka, and T. Moto-oka, "Highly parallel inference engine PIE—Goal rewriting model and machine architecture," *New Generation Computing*, vol. 2, no. 1, pp. 37-58, 1984.

[83] E. Goto, T. Ida, K. Hiraki, M. Suzuki, and N. Inada, "FLATS, A machine for numerical, symbolic and associative computing," in *Proc. 6th Int. Joint Conf. Artificial Intelligence*, pp. 1058-1066, Aug. 1979.

[84] N. Greenfield and A. Jericho, "A Professional's Personal Computer System," in *Proc. 8th Int. Symp. Computer Architecture*, pp. 217-226, 1981.

[85] M. Griss and M. Swanson, "MBALM/1700: A microprogrammed Lisp machine for the Burroughs B1726," in *Proc. MICRO-10*, 1977.

[86] A. Gupta, C. L. Forgy, A. Newell, and R. Wedig, "Parallel algorithms and architectures for rule-based systems," in *Proc. 13th Int. Symp. Computer Architecture*, pp. 28-37, 1986.

[87] J. Halpern and Y. Moses, "A guide to the modal logics of knowledge and belief: Preliminary draft," in *Proc. Int. Joint Conf. Artificial Intelligence*, pp. 480-490, 1985.

[88] R. H. Halstead, Jr., "Implementation of MULTILISP: LISP on multiprocessor," in *Proc. Symp. LISP and Functional Programming*, 1984.

[89] R. Halstead, "Parallel symbolic computing," *Computer*, vol. 19, no. 8, pp. 35-43, Aug. 1986.

[90] R. H. Halstead, Jr., "An assessment of Multilisp: Lessons from experience," *Int. J. Parallel Programming*, vol. 15, no. 6, pp. 459-501, Dec. 1986.

[91] R. Halstead, Jr., and J. Loaiza, "Expection handling in Multilisp," in *Proc. Int. Conf. Parallel Processing*, pp. 822-830, Aug. 1985.

[92] R. Halstead, Jr., T. Anderson, R. Osborne, and T. Sterlig, "Concert: Design of a multiprocessor development system," in *Proc. Int. Symp. Computer Architecture*, pp. 40-48, June 1986.

[93] P. B. Hawthorn and D. J. DeWitt, "Performance analysis of alternative database machine architectures," *IEEE Trans. Software Engineering*, vol. SE-8, no. 1, pp. 61-75, Jan. 1982.

[94] H. Hayashi, A. Hattori, and H. Akimoto, "ALPHA: A high-performance Lisp machine equipped with a new stack structure and garbage collection system," in *Proc. 10th Annual Int. Symp. Computer Architecture*, pp. 342-348, June 1983.

[95] B. Hayes-Roth, "A blackboard architecture for control," *Artificial Intelligence*, vol. 26, no. 3, pp. 251-321, July 1985.

[96] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, *Building Expert Systems*, Reading, MA: Addison-Wesley, 1983.

[97] P. Henderson, *Function Programming, Application and Implementation*, Englewood Cliffs, NJ: Prentice-Hall, 1980.

[98] C. Hewitt, "Viewing control structure as patterns of passing messages," *Artificial Intelligence*, vol. 8, no. 3, pp. 323-364, 1977.

[99] C. Hewitt, "The apiary network architecture for knowl-

edgeable systems," in *Conf. Record Lisp Conf.*, pp. 107-117, 1980.

[100] C. Hewitt and H. Lieberman, "Design issues in parallel architectures for artificial intelligence," in *Proc. IEEE COMPCON Spring*, pp. 418-423, Feb. 1984.

[101] C. E. Hewitt, *Description and Theoretical Analysis of PLANNER: A Language for Proving Theorems and Manipulating Models in Robot*, MIT AI Lab. TR-258, 1972.

[102] Y. Hibino, "A practical parallel garbage collection algorithm and its implementations," in *Proc. 7th Annual Symp. Computer Architecture*, pp. 113-120, May 1980.

[103] W. D. Hillis, *The Connection Machine*, Cambridge, MA: MIT Press, 1985.

[104] B. K. Hillyer and D. E. Shaw, "Execution of OPS5 production systems on a massively parallel machine," *J. Parallel Distributed Computing*, vol. 3, no. 2, pp. 236-268, 1986.

[105] J. Hintikka, "Impossible possible world vindicated," *J. Philosophical Logic*, pp. 475-484, 1975.

[106] G. Hinton, J. L. McClelland, and D. D. Rumelhart, "Distributed representations," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, D. E. Rumelhart, J. L. McClelland and the PDP Research Group ed., vol. 1, Cambridge, MA: MIT Press, 1986.

[107] G. Hinton and T. J. Sejnowski, "Learning and relearning in Boltzmann machines," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, D. E. Rumelhart, J. L. McClelland and the PDP Research Group ed., vol. 1, Cambridge, MA: MIT Press, 1986.

[108] G. E. Hinton, "Connectionist learning procedures," Tech. Rep. CMU-CS-87-115, Carnegie Mellon University, Pittsburgh, PA, June 1987.

[109] J. J. Hopfield and D. W. Tank, "Neural computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, no. 3, pp. 1-25, July 1985.

[110] J. J. Hopfield and D. W. Tank, *Disordered Systems and Biological Organization*, Springer-Verlag, 1986.

[111] J. J. Hopfield and D. W. Tank, "Computing with neural circuits: A model," *Science*, pp. 625-633, Aug. 1986.

[112] H. A. H. Ibrahim, J. R. Kender, and D. E. Shaw, "Low-level image analysis tasks on fine-grained tree-structured SIMD machines," *J. Parallel and Distributed Computing*, pp. 546-574, Dec. 1987.

[113] N. Ichiyoshi, T. Miyazaki and K. Taki, "A distributed implementation of flat GHC on the multi-PSI," presented at Int. Conf. on Logic Programming, 1987.

[114] N. Ito *et al.*, "The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D," in *Proc. 13th Int. Symp. Comput. Architecture*, pp. 149-156, 1986.

[115] N. Ito, H. Shimizu, M. Kishi, E. Kuno, and K. Rokusawa, "Dataflow based execution mechanisms of parallel and concurrent prolog," *New Generation Computing*, vol. 3, pp. 15-41, OHMSHA Ltd. and Springer-Verlag, 1985.

[116] P. Kanerva, "Parallel Structures in Human and Computer Memory," RIACS Tech. Rep. TR-86.2, NASA Ames Research Center, Moffett Field, CA, Jan. 1986.

[117] R. M. Keller, F. C. H. Lin, and J. Tanaka, "Rediflow multiprocessing," in *Proc. IEEE COMPCON Spring*, pp. 410-417, 1984.

[118] C. Kellogg, "Intelligent assistants for knowledge and information resources management," in *Proc. 8th Int. Joint Conf. on Artificial Intelligence*, pp. 170-172, Los Altos, CA, 1983.

[119] D. F. Kibler, "Parallelism in AI programs," in *Proc. Int. Joint Conf. on Artificial Intelligence*, pp. 53-56, 1985.

[120] M. King and M. Rosner, "Scanning the issue: The special issue on knowledge representation," *Proc. IEEE*, vol. 74, no. 10, pp. 1299-1303, Oct. 1986.

[121] J. de Kleer, "An assumption-based TMS," *Artificial Intelligence*, vol. 28, pp. 127-161, 1986.

[122] J. de Kleer, "Extending the ATMS," *Artificial Intelligence*, vol. 28, pp. 163-196, 1986.

[123] J. de Kleer, "Problem solving with the ATMS," *Artificial Intelligence*, vol. 28, pp. 197-224, 1986.

[124] T. Knight, "The CONS Microprocessor," AI Working paper 80, MIT, Cambridge, MA, Nov. 1974.

[125] R. Kowalski, "Predicate logic as a programming language," *IFIP Information Processing*, pp. 569-574, 1974.

[126] R. Kowalski, *Logic for Problem Solving*, North-Holland, 1979.

[127] H. Kung and S. Song, "An Efficient Parallel Garbage Collection Systems and Its Correctness Proof," Tech. Rep. Dept. Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, Sept. 1977.

[128] R. Kurzweil, "What is artificial intelligence anyway?" *American Scientist*, vol. 73, no. 3, pp. 258-264, May/June 1985.

[129] T. H. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *Comm. of the ACM*, vol. 27, no. 6, pp. 594-602, June 1984.

[130] L. E. Larson, J. F. Jensen, and P. T. Greiling, "GaAs high-speed digital IC technology: An overview," *Computer*, vol. 19, no. 10, pp. 21-28, Oct. 1986.

[131] T. F. Lehr and R. G. Wedig, "Toward a GaAs realization of a production-system machine," *Computer*, vol. 20, no. 4, pp. 36-49, Apr. 1987.

[132] D. B. Lenat and J. McDermott, "Less than general production system architectures," in *Proc. 5th Int. Joint Conf. on Artificial Intelligence*, pp. 923-932, Los Altos, CA, 1977.

[133] D. B. Lenat, "The ubiquity of discovery," in *Proc. 5th Int. Joint Conf. on Artificial Intelligence*, pp. 1093-1105, Los Altos, CA, 1977.

[134] D. B. Lenat, "The nature of heuristics," *Artificial Intelligence*, vol. 19, no. 2, pp. 189-249, 1982.

[135] D. B. Lenat, "Computer software for intelligent systems," *Scientific American*, vol. 251, no. 3, pp. 204-213, Sept. 1984.

[136] G. J. Li and B. W. Wah, "Computational efficiency of parallel approximate branch-and-bound algorithms," in *Proc. IEEE Int. Conf. on Parallel Processing*, pp. 473-480, Aug. 1984.

[137] G.-J. Li and B. W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Computers*, vol. C-34, no. 1, pp. 66-77, Jan. 1985.

[138] G.-J. Li and B. W. Wah, "MANIP-2: A multicomputer architecture for evaluating logic programs," in *Proc. IEEE Int. Conf. on Parallel Processing*, pp. 123-130, Aug. 1985. Also in *Tutorial: Computers for Artificial Intelligence Applications*, ed. B. W. Wah, IEEE Computer Society, 1986, pp. 392-399.

[139] B. W. Wah and G.-J. Li, "Multiprocessing of logic programs," in *Proc. IEEE Int. Conf. on Systems, Man and Cybernetics*, pp. 563-567, Oct. 1986.

[140] G. J. Li and B. W. Wah, "Optimal granularity of parallel evaluation of AND-trees," *Proc. Fall Joint Computer Conf.*, pp. 297-306, Nov. 1986.

[141] G. J. Li and B. W. Wah, "How good are parallel and ordered depth-first searches?" in *Proc. IEEE Int. Conf. on Parallel Processing*, pp. 992-999, Aug. 1986.

[142] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Comm. ACM*, vol. 26, no. 6, pp. 419-429, June 1983.

[143] G. Lindstrom and P. Panangaden, "Stream-based execution of logic programs," in *Proc. IEEE Int. Symp. on Logic Programming*, pp. 168-176, Feb. 1984.

[144] R. P. Lippmann, "An introduction to computing with neural nets," *IEEE Acoustics, Speech, and Signal Processing Mag.*, pp. 4-22, April 1987.

[145] F. J. Malabarba, "Review of available database machine technology," in *Proc. IEEE Trends and Appl.*, pp. 14-17, 1984.

[146] Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG: A new approach to logic programming," in *Logic Programming*, D. DeGroot and G. Lindstrom, Eds. Englewood Cliffs: Prentice-Hall, 1985.

[147] G. Matthews, R. Hewes, and S. Krueger, "Single-chip processor runs Lisp environments," *Computer Design*, pp. 69-76, May 1987.

[148] A. D. McAulay, "Spatial-light-modulator interconnected computers," *Computer*, vol. 20, no. 10, pp. 45-58, Oct. 1987.

[149] G. McCalla and N. Cercone (Eds.), "Special Issue on Knowledge Representation," *Computer*, vol. 16, no. 10, Oct. 1983.

[150] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, pt. I," *Comm. of the ACM*, vol. 3, no. 4, pp. 184-195, 1960.

[151] J. McCarthy and P. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," *Machine Intelligence 4*, pp. 463-502, 1969.

[152] J. McCarthy, "History of Lisp," *SIGPLAN Notices*, vol. 13, no. 8, pp. 217-223, 1978.

[153] J. L. McClelland, D. D. Rumelhart, and G. Hinton, "The appeal

of parallel distributed processing," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, D. E. Rumelhart, J. L. McClelland and the PDP Research Group, vol. 1. Cambridge, MA: MIT Press, 1986.

[154] J. F. McDonald, H. J. Greub, R. H. Steinvorth, B. J. Donlan, and A. S. Bergendahl, "Wafer scale interconnections for GaAs packaging—applications to RISC architecture," *Computer*, vol. 20, no. 4, pp. 21–35, Apr. 1987.

[155] J. R. McGraw, "Data flow computing: Software development," *IEEE Trans. Computers*, vol. C-29, no. 12, pp. 1095–1103, 1980.

[156] C. S. Mellish, "An alternative to structure sharing in the implementation of a prolog interpreter," in *Logic Programming*, ed. K. Clark and S. A. Tarnlund, Eds. New York, NY: Academic Press, 1982, pp. 99–106.

[157] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine Learning: An Artificial Intelligence Approach*, Tioga, 1983.

[158] M. Minsky, "A framework for representing knowledge," in *The Psychology of Computer Vision*, P. H. Winston, Ed. New York, NY: McGraw-Hill, 1975.

[159] D. I. Moldovan and Y. W. Tung, "SNAP: A VLSI architecture for artificial intelligence processing," *J. Parallel and Distributed Computing*, pp. 109–131, May 1985.

[160] L. Monier and P. Sidhu, "The architecture of the dragon," in *Proc. IEEE COMPCON*, pp. 118–121, Spring 1985.

[161] D. A. Moon, "Symbolics architecture," *Computer*, vol. 20, no. 1, pp. 43–52, Jan. 1987.

[162] T. Moto-oka, "Overview to the fifth generation computer system project," in *Proc. 10th Annual Int. Symp. on Computer Architecture*, pp. 417–422, June 1983.

[163] T. Moto-oka, H. Tanaka, H. Aida, K. Hirata, and T. Maruyama, "The architecture of a parallel inference engine (PIE)," in *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 479–488, 1984.

[164] T. Moto-oka and H. S. Stone, "Fifth-generation computer systems: A Japanese project," *Computer*, vol. 17, no. 3, pp. 6–13, Mar. 1984.

[165] A. Mukhopadhyay, "Hardware algorithms for nonnumeric computation," *IEEE Trans. Computers*, vol. C-28, no. 6, pp. 384–394, June 1979.

[166] K. Murakami, T. Kakuta, R. Onai, and N. Ito, "Research on parallel machine architecture for fifth-generation computer systems," *Computer*, vol. 18, no. 6, pp. 76–92, June 1985.

[167] K. Murakami, "Research on parallel machine architecture for fifth generation computing systems," *Computer*, p. 92, 1985.

[168] M. Nagao, J. I. Tsujii, K. Nakajima, K. Mitamura, and H. Ito, "Lisp machine NK3 and measurement of its performance," in *Proc. 6th Int. Joint Conf. on Artificial Intelligence*, pp. 625–627, Los Altos, CA, Aug. 1979.

[169] H. Nakagawa, "AND parallel prolog with divided assertion set," in *Proc. Symp. on Logic Programming*, pp. 22–28, 1984.

[170] R. Nakazaki, *et al.*, "Design of a high speed prolog machine (HPM)," in *Proc. 12th Int. Symp. on Computer Architecture*, pp. 191–197, 1985.

[171] P. M. Neches, "Hardware support for advanced data management systems," *Computer*, vol. 17, no. 11, pp. 29–40, Nov. 1984.

[172] A. Newell, J. C. Shaw, and H. A. Simon, "Programming the logic theory machine," in *Prof. IRE 1957 Western Joint Computer Conf.*, pp. 230–240, 1957.

[173] A. Newell and H. A. Simon, in *Human Problem Solving*. *Englewood Cliffs*: Prentice-Hall, 1972.

[174] A. Newell and H. Simon, "ACM Turing Award Lecture: Computer science as an empirical inquiry: Symbols and search," *Comm. ACM*, vol. 19, no. 3, Mar. 1975.

[175] A. Newell, "Production systems: Models of control structures," in *Visual Information Processing*, W. G. Chase, Ed. New York: Academic Press, 1975.

[176] H. P. Nii, "Blackboard systems, blackboard systems from a knowledge engineering perspective," *AI Mag.*, pp. 82–106, Aug. 1986.

[177] K. Oflazer, "Partitioning in parallel processing of production systems," in *Proc. IEEE Int. Conf. on Parallel Processing*, pp. 92–100, 1984.

[178] E. A. Ozkarahan, S. A. Schuster, and K. C. Smith, "RAP—An associative processor for database management," in *Proc. National Computer Conf.*, pp. 379–388, 1975.

[179] D. A. Patterson, "Reduced instruction set computers," *Comm. ACM*, vol. 28, no. 1, pp. 8–21, Jan. 1985.

[180] J. Pearl, *Heuristics—Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley, 1984.

[181] J. Pearl, "Some recent results in heuristic search theory," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 1, pp. 1–13, Jan. 1984.

[182] E. von Puttkamer, "A microprogrammed Lisp machine," *Microprocessing and Microprogramming*, vol. 11, no. 1, pp. 9–14, Jan. 1983.

[183] M. R. Quillian, "Word concepts: A theory and simulation of some basic semantic capabilities," *Behavioral Sci.*, pp. 410–430, 1967.

[184] C. V. Ramamoorthy, J. L. Turner, and B. W. Wah, "A design of a fast cellular associative memory for ordered retrieval," *IEEE Trans. Computers*, vol. C-27, no. 9, pp. 800–815, Sept. 1978.

[185] C. V. Ramamoorthy, A. Prakash, W.-T. Tsai, and Y. Usuda, "Software engineering," *Computer*, vol. 17, pp. 191–210, Oct. 1984.

[186] C. V. Ramamoorthy, S. Shekhar, and V. Garg, "Software development support for AI programs," *Computer*, pp. 30–42, Jan. 1987.

[187] U. S. Reddy, "On the relationship between logic and functional languages," in *Logic Programming*, D. DeGroot and E. G. Lindstrom Ed. Englewood Cliffs: Prentice-Hall, 1985.

[188] P. Robinson, "The SUM: an AI co-processor," *Byte*, pp. 169–180, June 1985.

[189] F. Rosenblatt, *Principles of Neurodynamics*. New York: Spartan Books, 1962.

[190] D. D. Rumelhart, G. Hinton, and J. L. McClelland, "A general framework for parallel distributed processing," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, ed. D. E. Rumelhart, J. L. McClelland and the PDP Research Group, vol. 1. Cambridge, MA: MIT Press, 1986.

[191] D. E. Rummelhart and J. L. McClelland *et al.* (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press, 1986.

[192] E. Sandewall, "An approach to the frame problem, and its implementation," *Machine Intelligence* 7, pp. 195–204, 1972.

[193] J. Sansonnet, D. Botella, and J. Perez, "Function distribution in a list-directed architecture," *Microprocessing and Microprogramming*, vol. 9, no. 3, pp. 143–153, 1982.

[194] J. P. Sansonnet, M. Castan, and C. Percebois, "M3L: A list-directed architecture," in *Proc. 7th Annual Symp. on Computer Architecture*, pp. 105–112, May 1980.

[195] J. P. Sansonnet, M. Castan, C. Percebois, D. Botella, and J. Perez, "Direct execution of Lisp on a list-directed architecture," in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, pp. 132–139, Mar. 1982.

[196] D. Schaefer and J. Fischer, "Beyond the supercomputer," *IEEE Spectrum*, vol. 19, no. 3, pp. 32–37, Mar. 1982.

[197] M. Schor, "Declarative knowledge programming: Better than procedural," *IEEE Expert*, vol. 1, no. 1, pp. 36–43, Spring 1986.

[198] T. J. Sejnowksi and C. R. Rosenberg, "NETtalk: a Parallel Network that Learns to Read Aloud," Tech. Rep., John Hopkins University, Baltimore, MD, Jan. 1986.

[199] G. Shafer, *A Mathematical Theory of Evidence*. Princeton, NJ: Princeton Univ. Press, 1976.

[200] E. Shapiro and A. Takeuchi, "Object oriented programming in concurrent prolog," *New Generation Computing*, vol. 1, no. 1, pp. 25–48, OHMSHA Ltd. and Springer-Verlag, 1983.

[201] S. M. Shatz and J.-P. Wang, "Introduction to distributed-software engineering," *Computer*, pp. 23–32, Oct. 1987.

[202] D. E. Shaw, "On the Range of Applicability of an Artificial Intelligence Machine," Tech. Rep., Columbia University, New York, NY, Jan. 1985.

[203] Y. Shih and K. B. Irani, "Large scale unification using a mesh-connected array of hardware unifers," in *Proc. Int. Conf. on Parallel Processing*, pp. 787–794, St. Charles, IL, 1987.

[204] Y. Shobatake and H. Asio, "A unification processor based on uniformly structured cellular hardware," in *Proc. 13th Int. Symp. on Computer Architecture*, pp. 140–148, 1986.

[205] H. A. Simon, "Search and reasoning in problem solving," *Artificial Intelligence*, vol. 21, pp. 7–29, 1983.

[206] H. A. Simon, "Whether software engineering needs to be artificially intelligent," *IEEE Trans. Software Engineering*, vol. SE-12, no. 7, July 1986.

[207] R. G. Smith, "The contract net protocol: High-level communication and control in a distributed problem solver," *IEEE Trans. Computers*, vol. C-29, no. 12, 1980.

[208] A. Snyder, "Object-Oriented Programming for Common Lisp," Rep. ATC-85-1, Software Technology Lab., Hewlett-Packard Lab., Palo Alto, CA, 1985.

[209] C. Stanfill and D. Waltz, "Toward memory-based reasoning," *Comm. ACM*, pp. 1213–1228, Dec. 1986.

[210] M. Stefik and D. G. Bobrow, "Object-oriented programming: Themes and variations," *AI Mag.*, pp. 40–62, Spring 1986.

[211] S. J. Stolfo and D. E. Shaw, "DADO: A Tree-Structured Machine Architecture for Production Systems," Tech. Rep., Columbia University, New York, NY, Mar. 1982.

[212] S. J. Stolfo and D. P. Miranker, "The DADO production system machine," *J. Parallel and Distributed Computing*, vol. 3, no. 2, pp. 269–296, 1986.

[213] S. J. Stolfo, "Initial performance of the DADO-2 prototype," *Computer*, pp. 75–84, Jan. 1987.

[214] H. S. Stone, "Parallel querying of large databases: A case study," *Computer*, pp. 11–21, Oct. 1987.

[215] S. Y. W. Su, "Associative programming in CASSM and its applications," in *Proc. 3rd Int. Conf. Very Large Databases*, pp. 213–228, 1977.

[216] P. A. Subrahmanyam and J-H You, "FUNLOG: A computational model integrating logic programming and functional programming," in *Logic Programming*, D. DeGroot and G. Lindstrom, Eds. Englewood Cliffs: Prentice-Hall, 1985.

[217] S. Sugimoto, K. Agusa, K. Tabata, and Y. Ohno, "A multimicroprocessor system for concurrent Lisp," in *Proc. IEEE Int. Conf. Parallel Processing*, pp. 135–143, 1983.

[218] G. J. Sussman and D. V. McDermott, "From PLANNER to CONNIVER—A genetic approach," *Fall Joint Computer Conf.*, vol. 41, pp. 129–137, 1972.

[219] G. J. Sussman, J. Holloway, G. L. Steel Jr., and A. Bell, "Scheme-79—Lisp on a chip," *Computer*, vol. 14, no. 7, pp. 10–21, July 1981.

[220] K. Takahasi, H. Yamada, H. Nagai, and K. Matsumi, "A new string search hardware architecture for VLSI," in *Proc. 13th Int. Symp. Computer Architecture*, pp. 20–27, 1986.

[221] A. Takeuchi and K. Fukukawa, "Parallel logic programming languages," in *Proc. Third Int. Conf. Logic Programming*, 1986.

[222] K. Taki, Y. Kaneda, and S. Maekawa, "The experimental Lisp machine," in *Proc. 6th Int. Joint Conf. on Artificial Intelligence*, pp. 865–867, Aug. 1979.

[223] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima, and A. Mitsuishi, "Hardware design and implementation of the personal sequential inference machine (PSI)," in *Proc. Int. Conf. Fifth Generation Computer Systems*, pp. 398–409, 1984.

[224] W. Teitelman and L. Masinter, "The Interlisp programming environment," *Computer*, vol. 14, no. 4, pp. 25–33, Apr. 1981.

[225] M. F. M. Tenorio and D. I. Moldovan, "Mapping production systems into multiprocessors," in *Proc. IEEE Int. Conf. Parallel Processing*, pp. 56–62, 1985.

[226] E. Tick and D. H. D. Warren, "Towards a pipelined prolog processor," *New Generation Computing*, vol. 2, no. 4, pp. 323–345, 1984.

[227] S. Torii *et al.*, "A database system architecture based on a vector processing method," in *Proc. 3rd Int. Conf. Data Engineering*, Feb. 1987.

[228] P. C. Treleaven and I. G. Lima, "Japan's fifth-generation computer systems," *Computer*, vol. 15, no. 8, pp. 79–88, Aug. 1982.

[229] A. B. Tucker, Jr., "A perspective on machine translation: Theory and practice," *Comm. ACM*, vol. 27, no. 4, pp. 322–329, Apr. 1984.

[230] S. Uchida, "Inference machines in FGCS project," in *Proc. VLSI'87 Int. Conf.*, IFIP TC-10, WG 10.5, Aug. 1985.

[231] K. Ueda, "Guarded Horn Clauses," Tech. Rep. TR-103, ICOT, Tokyo, Japan, 1985.

[232] L. M. Uhr, "Parallel-serial production systems," in *Proc. 6th Int. Joint Conf. Artificial Intelligence*, pp. 911–916, Aug. 1979.

[233] D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," *ACM SIGSOFT*, vol. 9, no. 3, pp. 157–167, May 1984.

[234] D. Ungar, R. Blau, P. Foley, D. Samples, and D. A. Patterson, "Architecture of SOAR: Smalltalk on RISC," in *Proc. 11th Ann. Int. Symp. Computer Architecture*, pp. 188–197, 1984.

[235] D. Ungar and D. Patterson, "What price Smalltalk?" *Computer*, pp. 67–74, Jan. 1987.

[236] S. R. Vegdahl, "A survey of proposed architectures for the execution of functional languages," *IEEE Trans. Computers*, vol. C-33, no. 12, pp. 1050–1071, Dec. 1984.

[237] J. S. Vitter and R. A. Simons, "New classes for parallel complexity: A study of unification and other complete problems," *IEEE Trans. Computers*, vol. C-35, no. 5, pp. 403–418, May 1986.

[238] B. W. Wah, G. J. Li, and C. F. Yu, "Multiprocessing of combinatorial search problems," *Computer*, vol. 18, no. 6, pp. 93–108, June 1985. Also in Tutorial: Computers for Artificial Intelligence Applications, ed. B. W. Wah, 1986, pp. 173–188.

[239] B. W. Wah, "Guest editor's introduction: New computers for artificial intelligence processing," *Computer*, vol. 20, no. 1, pp. 10–15, Jan. 1987.

[240] D. L. Waltz, "Applications of the connection machine," *Computer*, vol. 20, no. 1, Jan. 1987.

[241] D. H. Warren, L. M. Pereira, and F. Pereira, "Prolog—The language and its implementation compared with Lisp," in *Proc. Symp. Artificial Intelligence and Programming Languages, also SIGART Newsletter*, vol. 64, pp. 109–115, Aug. 1977.

[242] P. Wegner and B. Shriver (ed.), "Special issue on object-oriented programming workshop," *SIGPLAN Notices*, vol. 21, no. 10, Oct. 1986.

[243] M. Weiser, S. Kogge, M. McElvany, R. Pierson, R. Post, and A. Thareja, "Status and performance of the ZMOB parallel processing system," in *Proc. IEEE COMPCON Spring*, pp. 71–73, Feb. 1985.

[244] L. C. West, "Picosecond integrated optical logic," *Computer*, vol. 20, pp. 34–47, Dec. 1987.

[245] T. Winograd, "Frame representations and the declarative procedural controversary," in *Representation and Understanding: Studies in Cognitive Science*, pp. 185–210, Academic Press, Inc., 1975.

[246] P. H. Winston and B. Horn, *Lisp*, Second Edition, Reading, MA: Addison-Wesley, 1984.

[247] Y. Yamaguchi, K. Toda, and T. Yuba, "A performance evaluation of a Lisp-based data-driven machine (EM-3)," in *Proc. 10th Annual Int. Symp. Computer Architecture*, pp. 363–369, June 1983.

[248] Y. Yamaguchi, K. Toda, J. Herath, and T. Yuba, "EM-3: A Lisp-based data-driven machine," in *Proc. Int. Conf. Fifth Generation Computer Systems*, pp. 524–532, 1984.

[249] C. F. Yu, "Efficient Combinatorial Search Algorithms," Ph.D. Thesis, School of Electrical Engineering, Purdue University, West Lafayette, IN, Dec. 1986.

[250] L. A. Zadeh, "Fuzzy sets," *Information and Control*, pp. 338–353, 1965.

[251] L. A. Zadeh, "Approximate Reasoning Based on Fuzzy Logic," in *Proc. 6th Int. Joint Conf. Artificial Intelligence*, pp. 1004–1010, Aug. 1979.

**Benjamin W. Wah** (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of California, Berkeley, in 1979. He was on the faculty of the School of Electrical Engineering at Purdue University, West Lafayette, IN, between 1979 and 1985. He is now an Associate Professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign, Urbana. He is currently on leave at the National Science Foundation as a Program Director in the Microelectronic Information Processing Systems Division. His areas of research include computer architecture, parallel processing, artificial intelligence, distributed databases, and computer networks.

Dr. Wah is the Associate Editor-in-Chief of the forthcoming *IEEE Transactions on Knowledge and Data Engineering*, an area editor of

l SIG-

erson,
. *11th*
1984.
*Com-*

or the
*puters,*

l com-
prob-
03–418,

of com-
. 6, pp.
rtificial
73–188.
nputers
20, no.

chine,"

g—The
isp," in
*ng Lan-*
15, Aug.

ject-ori-
, vol. 21,

ost, and
parallel
, pp. 71-

;" *Com-*

clarative
*d Under-*
210, Aca-

Reading,

nce eval-
," in *Proc.*
. 363–369,

-3: A Lisp-
*Fifth Gen-*

ns," Ph.D.
University,

*l*, pp. 338-

zzy Logic,"
, pp. 1004-

ber, IEEE)
nputer sci-
California,
faculty of
ing at Pur-
N, between
ociate Pro-
ctrical and
e Coordi-
University
n, Urbana.
dation as a
Processing
puter archi-
buted data-

coming *IEEE*
ea editor of

the *Journal of Parallel and Distributed Computing*, and an editor of *Information Sciences*. He serves as a member of the Governing Board of the IEEE Computer Society and a program evaluator for ABET (computer Engineering) and CSAC (computer science). Previously, he served as chairman and member of program committee of a number of international conferences, an editor of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, and a distinguished visitor of the IEEE Computer Society.

Matthew B. Lowrie received the B.S. degree (highest honors) in computer engineering and the M.S. degree in electrical engineering from the University of Illinois, Urbana, in 1985 and 1986, respectively. He is currently a Ph.D. candidate in electrical engineering at the University of Illinois.

His research interests include artificial intelligence, learning, search algorithms, and fault tolerant computing. Mr. Lowrie is supported by a fellowship with AT&T Bell Laboratories.

Guo-jie Li graduated from Peking Unive sity in 1968 and received the M.S. degree computer science and engineering from tl University of Science and Technology China and the Institute of Computing Tec nology, Academia Sinica, in 1981. I received the Ph.D. degree in electrical en neering in 1985 from Purdue Universi Between 1985 and 1986, he was a post-dc toral research associate in the Coordinat Science Laboratory of the University of I nois at Urbana-Champaign.

Guo-jie Li is an Associate Professor at the Institute of Computi Technology, Academia Sinica. His research interests include p allel processing, artificial intelligence, theory of algorithms, a computer architecture. He has published over 30 papers and I co-edited *Tutorial: Computers for Artificial Intelligence Appli tions* (IEEE Computer Society Press, 1986). He has also written s eral chapters in several published books. He is an editor of the *Jo nal of Computer Science and Technology* and the *Journal of Patt Recognition and Artificial Intelligence*.