The Knowledge Engineering Review

Metalogic machines: a retrospective rationale for the Japanese Fifth Generation

G. A. RINGWOOD
Department of Computing, Imperial College, London SW7 2BZ, UK

Abstract

The oft quoted inadequacy of von Neumann architectures for AI applications has regularly been used to justify the design of special purpose parallel machines. In particular, the von Neumann computational model has been criticized as being unsuitable for parallelism because of the memory access bottleneck. For the design of a new machine both top-down and bottom-up methodologies have drawbacks. The middle-out strategy, working both up and down from an intrinsically concurrent high-level programming language as a means of both representing and processing knowledge provides an attractive way of providing a symbiosis between software and hardware. The longest established and most well-founded symbolic method for the representation and manipulation of knowledge is logic. A notable result of the last decade, work on mechanical theorem proving was that a subset of predicate logic, Horn Clauses, can form the foundation of a computational model. The execution model of Prolog, the first popular language based on Horn Clauses, was designed for efficient evaluation on von Neumann architectures. An alternative computational model, more suitable for expressing reactive systems but retaining Prolog's affinity for metaprogramming, has given rise to a new class of languages, concurrent logic languages. One among many of these languages, FGHC (flat Guarded Horn Clauses), was developed by the Japanese as the kernel of their Fifth Generation initiative. A background familiarity with Prolog would be helpful in understanding this article.

1 Introduction

The military instigation of computers towards the end of the Second World War through scientific use in the 1950s, commercial applications in the 1960s industrial applications in the 1970s and domestic consumer takeup in the 1980s plots the radius of expansion of the computing universe. Science fiction writers have speculated that intelligent machines will be a future generation of wealth creating consumer products. But, speculation in the area of artificial intelligence is far more ancient than the oft quoted foundations of computing in the numerical calculating machines of Babbage. One documented example is to be found in Aristotle's Politics written in the fourth century B.C. Aristotle classifies the slave as "an animate article of property" and suggests that slaves or subordinates in general might not be necessary if "each instrument could do its own work at command or by intelligent anticipation like the statues of Daedalus, or the tripods of Hephaestus". This reference to the legendary robots devised by the mythical technocrats Daedalus and Hephaestus, the former an artificer who made wings for Icarus and the latter a blacksmith god, testify to the proposition that the concept of robot, if not the name, was ancient even in Aristotle's time. The Japanese 5th Generation initiative is the present incarnation of this same aspiration. It is interesting to note that Aristotle concluded that even if such machines existed, human slaves would still be necessary to render the *personal services* without which it would be impossible to live well.

There are, today, two problematic parts to artificial intelligence. The first, the "artificial" bit and

the second the "intelligence" bit. The "artificial" part appears simple by comparison with the difficulty of defining the concept of "intelligence". The problem can be clarified, somewhat, by an awareness that the combination "artificial intelligence" is used in two not altogether distinct connotations. The first, which tries to make computers in the human image and the second which tries to build computers to perform tasks that most people would agree appears to require human intelligence. The first nicely sidesteps the question of the definition of intelligence. There is, of course, the non-constructive point of view that anything done by a machine cannot a priori be intelligent. The difficulty with this philosophy is that as machines become more and more capable, humans become less and less intelligent. This attitude can be likened to the reaction of religious dogmatists to Darwinian theories or the even earlier Copernican revolution.

The first approach to AI has led to machine architectures patterned after the neural networks of the human brain. These designs result in massively parallel computers with thousands of processors. Each processor has a small local storage and functions on an extremely reduced instruction set. In neural networks there is no stored database; no carefully worked out program. The only principle that guides the machine is that it incorporates a notion of right and wrong and is constructed so as to strive for correctness. In this way the machine can be self-learning; each input to the machine produces an output, correct outputs are reinforcing, incorrect outputs cause internal reconfiguration. By adjusting its internal communication network the machine strives to achieve approved responses. At first response is by blind trial and error; later on, as the learning process continues, it becomes a mixture of trial and error and experience. Eventually the machine behaves as if it "knew" exactly what it was the instructor (she cannot really be called a programmer) was trying to tell it.

By the time the neural machine has learned something, the instructor does not know at the conceptual level what is going on inside the machine—it is far too complex for that. This is not an entirely satisfactory state of affairs as the bulk of AI is concerned with explanations of why things are the way they are. There is no way of justifying a neural machine's basic responses, because they are conditioned. Given that these machines are fashioned in the human image they are likely to be as fallible as humans are. One of the primary aspects of AI is sound reasoned arguments to any conclusion. Gut reactions and guesswork may be indicative but cannot be relied upon. One of the attractions that the neural machine has however, is robustness to noisy and errant input data.

Current thinking on human perception is that we continuously relate fragmentary stimuli to knowledge familiar from various experiences and we unconsciously test and reiterate our perceptions at different levels of abstraction. In other words, what we believe we perceive is in fact only a mental reconstruction made up from fragments of sensory data. This is not too far removed from the philosophies of Husserl and Heidegger and perhaps philosophical logic has a role to play in the design of intelligent machines. If this is the case, it is not an either/or situation of symbolic concepts versus artificial neurons but a complementary one. They are different levels of abstraction, with the symbolic approach forming a higher level, and hybrid machines could have a role to play in a truly intelligent machine. Pure neural machines have, however, been proposed as the Sixth Generation before the Fifth Generation has even got off the ground.

It is a fallacy that in order for a machine to behave as if it were intelligent it would have to function in the same way as the human brain. That this is fallacious is attested to by the fact that the computers are much better at number-crunching than humans are, yet they do not perform arithmetic operations in the same way. ENIAC was the first and only well known electronic computer to employ a base 10 number system. This departure of machine intelligence from human intelligence is not necessarily a drawback but can be advantageous; machines can do with patience some things which humans can't, such as exhaustive searches and the efficient transfer of information. The human child does not, as far as has been established, inherit the knowledge of its parents but has to undergo the same painful learning process from scratch just as its parents had done. On the other hand machine knowledge bases can be preserved, duplicated, merged and shared between machines without any loss. A machine with access to all knowledge is not inconceivable. The Fifth Generation has pursued the second meaning of AI but because of difficulty in definition it will only be distinguished by the fact that the symbolic representation and processing of knowledge are central to it.

2 Symbolic AI machines

AI involves the symbolic encoding and processing of knowledge about objects, relations, goals, actions and processes. Prominent among the symbolic approaches to knowledge representation are semantic nets, frames, production systems, objects, logic and relational databases. Processing of knowledge is required for problem solving, reasoning and information retrieval. Typical methods of computation are deduction for logic, forward chaining for production systems, marker propagation for semantic networks and procedural attachments for frames. These computations involve intensive memory accesses rather than ALU operations. The characteristics of symbolic processing are, thus, fundamentally different from numerical requirements. Unfortunately, many architectural features of conventional computers are tuned for numerical applications. It is claimed that a von Neumann architecture with centralized control presents a processor-memory bottleneck (a channel through which all communication must flow) to the intensive and irregular memory access patterns that are typical of AI applications. Despite the drawbacks of non-accountability, neural machines appear to offer an advance over the von Neumann architecture in that they do not have such a bottleneck and much more silicon can be active at any one time; neural machines are intrinsically highly parallel.

In the realm of numerical processing, it appears that when it comes to performing highly specialized computations then, dollar for dollar, a tailor-made computer can easily outstrip the biggest mainframe despite their noted affinity for numerical calculation. A homemade computer, the Dubner's machine (Devlin, 1987) (constructed by a father and son in the US) at the time of the article held a number of world records for natural number computation, yet the components of the machine only cost around $1000. According to Devlin, this machine had found: the largest known palindromic prime, the largest prime of the form N! + 1, the largest known prime-factorial plus 1 and a number of other world records of a similiar nature. For what it was capable of, the Dubner's machine was claimed at the time to be only ten times slower than a Cray-1 and the equal of anything that came out of IBM.

The fundamental inefficiency of the von Neumann machine for AI processing is the memory, in which only a single location can change state at a given time. Any machine with a centralized decision making mechanism would inevitably become a bottleneck. The achievements of the Dubner's machine and neural net machines encourage the speculative design of special purpose parallel machines for AI applications. The purpose of this paper is to expose the reader to a rationale (albeit with a great deal of hindsight) behind the Japanese vision of the Fifth Generation in terms of the pursuit of the perfect "AI machine". (In the wake of the Japanese initiative there have been a number of other visions of the Fifth Generation which differ, some fundamentally, from the Japanese one; these will not be discussed.) This does not mean that the arguments in favour of this approach are compelling, original or even consistent but taken together they hold some fascination.

The original Japanese proposal was to base the machine design not around hardware or applications, but a programming language—Prolog. Arguments as to why this may have come about are presented in the paper in a somewhat anecdotal form, since this is the nature of faith. The Japanese quickly realized that the computational model of Prolog is very much geared to the von Neumann computational model. It is not particularly well suited to either parallel hardware, which in view of the success of organic neural nets can be regarded as a possible solution to the bottleneck, or to one of the most critical programming tasks, reactive systems. Operating systems and environments are examples of reactive programs. This resulted in two feedback loops of design modification to the chosen language. One loop was driven by the desire for efficient implementation; the other by the need for ease of expression. In this way hardware constraints were not achieved at the expense of applications. This iterative design process has led to the development of a new family of languages, the concurrent logic languages one of which, FGHC, Flat Guarded Horn Clauses, is the kernel language of the Japanese Fifth Generation initiative. The development of these languages has been accompanied by a change in attitude towards Horn Clause Logic (the theoretical basis of Prolog). For concurrent logic languages, Horn Clause Logic is not primarily viewed as an object-level infer-

ence system, as is usual for Prolog, but rather as a metalevel inference system. This explains the title of the paper. A background familiarity with Prolog would help understanding the criticisms of it that are expressed in the article.

3 A design decision

The biggest problem with daunting tasks like how to design the ultimate AI machine or how to find the answer to life, the universe and everything, is where to start. The concept of a specialist AI machine extends across the boundaries of at least four computer disciplines: hardware; software; theory and, of course, applications. The boundary between software and hardware is bridged by hardware support for high-level AI application programming languages. The boundary between software and theory is crossed by language semantics and the complexity of structure is possible to express naturally in a language. (Natural here means without encoding another computational model.) Any of these disciplines could be, and probably have been, possible starting points for a machine design.

The design of an AI machine could begin bottom-up from what might be thought useful hardware. For example, it might be speculated that for artificial intelligence applications, content addressable store would be promising hardware if it is the opinion that it is the memory access of the von Neumann machine that is causing the bottleneck. This form of development starts from available hardware devices and works upwards towards the applications. Valuable lessons can be learned from what seem to be smart ideas in hardware that do not in practice contribute to a more efficient machine. Ungar and Patterson (1987) describe the salutary tale of the hardware "architect's trap" in designing support for a Smalltalk, an object-oriented language, machine:

- each idea was "clever";
- each idea made a particular operation much faster;
- not one idea significantly improved overall performance!

"The primary impact of *clever* ideas was to increase the difficulty of building the machine and thus lengthen the development cycle". This warning encourages an occamist's approach to design.

One danger in bottom-up methodology is then, "clever" hardware without a computational model to take best advantage of it. Bottom-up design involves specifying, refining and validating a computational model which the chosen hardware can support; the process is iterative. The von Neumann computer architecture can be seen to be have shaped computer science as it is today; imperative languages can be recognized as natural consequences of the von Neumann architecture. In a von Neumann machine the store has individually addressable locations. The effect of machine code operations is to change the contents of store locations; a variable in an imperative programming language can be regarded as an abstraction of the von Neumann computer store location. The accessing of consecutive memory locations in conventional machines naturally gives rise to the abstraction of a stack. This is reflected in block structured imperative languages with the consequent hierarchical calling structure of subroutines and variable scoping.

In a stack based computation only the top of stack is active. Highly parallel, machines in which large amounts of silicon are required to be in flux at any one time seem to be so conceptually different from stack based languages that it seems unlikely that extensions of the von Neumann computational model would prove suitable for programming them. With decreasing costs of hardware, radically different designs become feasible and multiprocessor systems could become the norm. There is no reason why the ratio of store locations to processors should not decrease and it is possible to conceive of machines with as many processors as store locations; then the von Neumann concept of store location disappears and with it, the bottleneck. An alternative computational model for AI with large scale parallelism is seen in Open Systems advocated by Hewitt (1985). The argument goes as follows: IF to avoid the von Neumann bottleneck, decision making is decentralized, no system part can directly control the resources of any other system part (otherwise it would itself become a bottleneck). The various autonomous parts must necessarily communicate with one

another if anything is to be cooperatively achieved. Possible communication delays or breakdowns indicate that inconsistent knowledge must be tolerated. According to Hewitt, incomplete knowledge requires an approach which allows continuous acquisition, refinement and toleration of inconsistency. The computational model for Open Systems is seen in Actors (Agha, 1986) another example of the Object-Oriented programming paradigm.

Using currently available hardware is not the only way to design a computer. As the von Neumann experience shows the choice of hardware dictates the computational models it can support efficiently. A more sensible proposition, perhaps, would be to try to design the hardware to serve the computational model, rather than the other way around.

4 The natue of AI applications

The antithesis of bottom-up design is a top-down methodology: a goal directed approach working from the applications down to the hardware. In principle, the result of bottom-up design should produce the same effect as top-down design. This, however, is rarely the case in practice. The large gap between the hardware and the application forces compromises, but the compromises take place at different stages in the two design methodologies. For example, tradeoffs must be performed as regards the amount of memory required to store the knowledge, time allowed for processing knowledge, the expected use of knowledge and the physical limitations of the underlying hardware. Rules can encapsulate large amounts of facts but require time for making inferences to reproduce these facts. Previously it was argued that one way to increase computational power for AI applications is to build special purpose machines. But there is danger in going too far and being too application specific; for example the applications might be limited to image analysis or speech recognition. In the confines of sensory perception, it might be argued that what is needed are specialized peripheral processors.

Top-down design involves specifying, refining and validating the requirements of the application. The role of the computer architect lies in choosing representations of knowledge, recognizing tasks for maintaining and learning knowledge, identifying primitive operations in the knowledge representation schemes and supporting these tasks in hardware and software. The process is iterative but much of the top-down refinement has already been embodied in knowledge representation systems. Top-down design requires an appreciation of the nature of AI applications. Having said that the problem of machine intelligence is not, necessarily, to model human intelligence, one still needs analogies and examples from human intelligence to guide the direction of research. An AI machine needs to represent reasoning and innovative thinking.

The nature of AI computation is very different from numerical computation for which general-purpose machines tend to be tuned. Procedures, like gaussian elimination, are algorithms about which the properties of efficiency and correctness can be rigorously argued. In contrast, AI is mainly concerned with empiric procedures. In fact a large part of the problem is to obtain a formal understanding of various heuristic procedures used in obtaining acceptable solutions to NP-hard (generic non-polynomial) problems. Numerical problems usually have a known set of inputs and a known set of outputs. They lend themselves to a structured program design where variable entities are known prior to subsequent detailed study of the problem definition.

Structured programming concepts have evolved in response to the ever increasing complexity demanded from software. The top-down development of software advocated by structured programming is highly appropriate for problems for which there is a predefined and predictable set of conclusions. Note that we are now talking about top-down software development rather than the top-down design approach to the design of the ultimate AI machine (I jest). However, the points that come from a discussion of top-down software development do have ramifications for general top-down design methodology and AI application programmming.

A problem can only be solved top-down if all objects and their interrelationship are known in advance. Pascal programmers are chided if they start coding before they understand the problem completely; they are expected to know all the data objects and data flows before they begin coding.

Top-down design tends to result in inherently sequential programs where control is subordinated to procedures and functions, in the recursive refinement process. Most real world problems which human beings encounter and manage successfully, consciously or unconsciously, do not fall into this category. The derivation of a solution does not follow an easily predictable path. These problems have a high degree of uncertainty as to the form of the solution. Such problem solving feats are generally accomplished as a result of our interaction with our environment and are not easily classified as algorithimic. We make decisions by a correlation of previous solutions to problems, taught experience and a set of heuristics. A heuristic is a rule-of thumb, strategy, trick, simplification or in general any device which drastically limits exhaustive search for solutions. That is, we break these problems down by using rules which allow us to arrive at intermediate problems from which we can achieve conclusive solutions. We also demand some form of explanation articulating the reasoning, inferences, which led to the particular solution.

In real world situations what we have is a problem of coping with complexity. One of the principal devices in finding solutions to complex systems is abstraction. We exclude factors in the real world which appear to have no or minimal effect on the solution to the problem at hand. In this way we model the world with a closed system which contains only those factors we consider relevant. An initial "attempt" at solution can be made based on a grossly oversimplified set of rules. In a rule based approach we can continually improve performance by adding rules and making modifications to existing rules. Continual, or incremental development takes place on the minimal system until it performs adequately. One of the characteristics of such systems is that they are never finished. Once an initial, but fairly limited system is operational, new rules can be added at any time to increase sophistication or scope. That is, we allow an expansion of the closed system to incorporate more factors. This is a flexibility that the top-down approach to software design does not have but is required in dealing with incomplete data. Such change in top-down structured system design generally leads to a complete redesign of the program. From a programming language point of view this style of programming is more suited to declarative languages than procedural ones since the integrity of the software is incrementally maintained by the language semantics and control is generally relegated to the run-time system.

AI problems are characterized by lack of knowledge and incomplete understanding. To solve such problems seems to require non-determinate behaviour; sometimes there appears to be a need to perform exploratory search of a number of possibilities. That is, at certain points in the computation there are choice points and it is not clear which is the best to choose. In general, once initialized, the choice points in a non-determinate computation give rise to independent non-interacting subtasks and this could be exploited by a parallel machine; this form of parallelism is known as OR-parallelism. However, there is a problem with OR-parallelism: when it is used recursively the generated parallel activities grow exponentially with respect to the number of levels in the search tree. Exponentially created activities are beyond the capabilities of any practical processing system. Without appropriately directing any search, restructuring computation on the fly and detecting redundant computations, much of the power of parallelism could be wasted. Owing to the lack of complete knowledge and uncertainty as to the form of the solution, computational resources have to be allocated and de-allocated dynamically; so it is impossible to plan in advance which procedures need to be run. There is, thus, a great need to take into account the organizational principles of knowledge.

At least two levels of knowledge can be distinguished: object-level and metalevel. Object knowledge refers to the relations between objects, events and actions per se while metaknowledge is knowledge about object knowledge. The object-level representation of knowledge is an important element in the design process and dictates whether a given problem can be solved in a reasonable amount of space and time. In AI applications, such as expert systems, a lot of reasoning is at the metalevel: Is this rule applicable to this situation? Is this theorem provable from these rules? How can one arrive at that conclusion? What would be the effect if this information were replaced by some other? In fact, metaknowledge exists in a hierarchy above object knowledge and can give information on the appropriate domain of object knowledge to apply, and the appropriate time to

apply it. Many knowledge representation schemes and program paradigms such as logic, frame, semantic networks, object-oriented and constraint systems can be integrated with the aid of metaknowledge. Metaknowledge allows one to express partial specifications of problem solving procedures making problem solvers more aesthetic, simpler to build and easier to modify. For example, always try this rule in preference to some other rule. Thus, metaknowledge, in the form of heuristics has potential for controlling the combinatorial search problem. Moreover, metaknowledge facilitates incremental system development; that is, one can start from a search intensive procedure and incrementally add control information until one obtains a search free algorithm.

5 Logic machines

A schema for knowledge representation influences both knowledge acquisition and processing, so should not be chosen without careful consideration of its repercussions. There are still many areas where, as yet, there are no totally satisfactory ways of symbolically representing common sense reasoning. The problem is essentially that of developing a formal semantics for reasoning in natural language. What AI adds to this enterprise is a concern for computational tractability; there is another large gap between what can be represented symbolically and what can be reasoned about efficiently. The most serious danger of the top-down design is an all embracing method of representing knowledge for which the associated processing is woefully slow or runs out of space on any architecture.

Technological progress appears to be made by the gradual refinement and new applications of well established theories rather than the punctuations of revolutionary activity in which new paradigms are established. If the various branches of discovery were to be measured by their relative antiquities, then the formal representation of knowledge by logic must be one of the most respectable. In Plato's time geometers speculated about machines which could support formal derivations in logic. According to Gardner (1982), the inventor of the world's first logic machine was the British statesman Charles Stanhope (1753–1816), third Earl of Stanhope. Stanhope was a prolific inventor; his logic machine, of the form of a square slide rule, was known as the Stanhope Demonstrator since it demonstrated symbolically the inferences which follow from syllogisms in a manner similar to Venn diagrams (which the Earl's invention anticipated). No description of the machine was published until 63 years after Stanhope's death; but another of his inventions an improved calculator, may have had more influence on computer science as two of these machines came into the possession of Babbage.

In the electronic era, the connection between logic and AI goes back to the earliest days. At the first formal conference on AI, held at Dartmouth College, Newell, Shaw and Simon (1956) presented the Logic Theorist, a heuristic theorem proving program for propositional logic which could prove some theorems from principia mathematica. These early days did not give logic a central role in knowledge representation, rather it was viewed as an example of programming a computer to perform a task that had previously been regarded as requiring human intelligence. The idea that logic can play a fundamental role grew, largely, out of the work of McCarthy.

In 1958, McCarthy published a paper, the central theme of which was that first order predicate calculus promised to be a universal language for representing common sense knowledge. In this proposal, a computer system is needed that can perform deductions from axioms representing the knowledge in order to solve everyday problems. However, at the time there did not exist sufficiently efficient algorithms for drawing inferences in predicate logic.

Early attempts to automate logical systems had foundered on the fact that they require instantiating variables at points in the search for a proof where insufficient information was available to justify any choice. This changed in 1965 with Robinson's exposition of the most general unifier (Robinson, 1965). The most general unifier (mgu) algorithm, a generalization of Herbrand's unification algorithm (Herbrand, 1930), enables a theorem prover to postpone choosing instances for quantified variables until dictated by the facts. McCarthy's ideas for using logic to represent commonsense knowledge and Robinson's most general unifier for automatic theorem proving were first

brought together by Green (1969) when he implemented a question answering cum problem solving system that used a resolution theorem prover as its inference engine.

However, the mgu did not solve all the problems. There was still a difficulty in that the search space grows so fast that problems of even moderate complexity could not be solved within reasonable time and space bounds. There were intolerable redundancies in the theorem prover which compounded the combinatorially explosive nature inherent in the problems. Various restrictions on resolution were investigated by many authors in the late 1960s and early 1970s which eliminated some redundancies. Notable among these restrictions were backward chaining (Loveland, 1969), linear resolution (Loveland, 1970) and input refutation (Chang, 1970).

These difficulties led to a period in the 1970s when logic was out of fashion and alternative, ad hoc, representational formalisms were developed. Logic is now, however, back in the forefront of computing research. It has been claimed that logic provides the theoretical framework that comes closest to doing for computing what continuum mathematics has done and continues to do for physics. The reason for this turn around of fortune lies in several developments: Firstly, drawing on the techniques of philosophical logic, rational reconstructions of the alternative knowledge representations systems have been developed in logic, particularly in belief systems, non-monotonic and default reasoning. These reconstructions have deepened understanding of both the problems and the solutions and thus provide a unifying formalism. Secondly, there has been much work in expert systems indicating that the efficiency problems of the early automatic theorem provers are not inevitable, but can be tempered by theory-specific (meta) control of the deduction process. Thirdly, mathematical theorem proving systems have been developed to the point where they now can be considered seriously as tools for software or hardware verification and synthesis. In particular, Green's ideas can be made to work by putting together backwards chaining, input refutation and restricting attention to a subset of predicate calculus called Horn Clause Logic (Kuehner, 1972, see also Ringwood 1988b, 1989 for a more detailed discussion of this point).

With hindsight the first logic programming language based on Horn Clauses was Absys (see, Elcock, 1988). The Absys view of unification is an algorithm for solving systems of simultaneous equations, called constraints. Constraints are partial finite specifications of (possibly infinite) sets of values for variables, which stand for fixed but possibly unknown objects in some domain of discourse. The fundamental idea in constraint programming is producing finer and finer upper bounds to the values of the variables. At every step of the computation, a defined relationship (goal) is replaced by its definition (body of a Horn Clause). In this process, new constraints and new variables may be introduced, thus further constraining the initial set of variables. As with the most general unifier, this avoids choosing values for unknowns until uniquely determined by the constraints. This approach was later taken up by Colmerauer (1982) and more recently by Jaffar and Lassez (1987).

However, logic programming was first popularized by Prolog (**Pro**grammation en **Log**ique, the first implementation was reported by Colmerauer (1972)) and this language has come to be viewed by many (mainly Europeans and Japanese) as the most powerful existing language for implementing many small designs for natural language systems, database systems, knowledge-based expert systems and other AI applications. The popularity of Prolog for AI applications stems from its unique nature, which combines simplicity of syntax, expressive and computational power allowing rapid prototyping, reasonably efficient implementation and simple, clean semantics. It has the unique property that its semantics are intuitively obvious. This is remarkable when you consider other programming languages where semantics require a great deal of mathematical prerequisites before they can begin to be understood.

6 The middle way

The large gap between hardware and applications and the need for compromises suggests that another approach to the design of an AI machine might be to work both upwards and downwards from some middle point. This strategy offers the potential for some synergy between hardware and applications. Much of bottom-up design has been encapsulated in computational models and much

of top-down design has been encapsulated in knowledge representation systems. The middle-out philosophy starts from a knowledge representation scheme and computational model suitable for representing and manipulating knowledge, and develops in both directions: upwards to the application and downwards to the hardware. In general, high-level programming languages provide both a system for representing knowledge and a computational model. LISP machines are an example of computers designed with a programming language as the middle layer. The earliest example of high-level language hardware support was on the PDP-6 and its successors. The half-word and stack instructions were developed with LISP in mind. The Burrough's 6000 series was specifically designed to support block structured languages, Algol in particular. More recent examples are the Inmos Transputer based on CSP (Hoare, 1985) and of course, the Japanese Fifth-Generation computer project based on Prolog. In these latter cases the starting language has been modified as a result of compromise between the computational and the hardware design issues. In the case of the Japanese, Prolog has been superceded by Flat Guarded Horn Clauses (FGHC) and in the case of Inmos, CSP by Occam (Inmos, 1984). The name CSP (Communicating Sequential Processes) (Hoare, 1985) is more properly reserved for a computational model rather than a programming language. The Transputer is significant in another respect; it is a language based microprocessor design rather than a complete machine design. In view of the fact that it is claimed that most new computers are based on off-the-shelf microprocessors (Schofield, 1988), an upward compatible microprocessor family seems a preferable approach to the design of a language based machine since it still leaves a great degree of design freedom.

General-purposes machines evolve more rapidly than special purpose machines because of larger markets. The high cost of designing special purpose machines such as LISP machines and the low sales usually means that manufacturers are reluctant to redesign them to take account of new developments in hardware and software. This means that old special purpose designs compare unfavourably with the most recent general purpose technology. For example, LISP now runs faster on RISC (Reduced Instruction Set Computers) machines than it does on outdated special purpose machines such as Xerox workstations. Certainly RISC machines can provide a better base for AI applications than CISC (Complex Instruction Set Computers) von Neumann machines because of the smaller granularity of the atomic computation step. The composition of these steps can be better tuned to the needs of AI than the large grain steps generally provided by CISC machines which tend to favour numerical applications.

Leaving aside this, perhaps, indefensible drawback, various disputes over the most appropriate language for AI programming have never been satisfactorily settled and split the AI community. An old chestnut of the 1960s is the declarative versus procedural representation of knowledge. Procedural schemes allow direct interaction of facts and heuristic information, hence eliminating wasteful searching. However, procedural languages are in general strongly influenced by the von Neumann architecture (e.g. block structured) and as a result tend to overspecify computational precedence and so constrain the degree of parallelism. The departure from procedural languages to declarative ones likens the departure from algorithms to heuristics. Declarative languages offer simpler semantics, clarity and conciseness and superficially appear to offer a higher potential for parallelism than procedural representations because they specify what and not how. Clean semantics is claimed to make application programming easier and more secure which is particularly important when software is becoming increasingly complex as more "intelligence" is provided.

Within the declarative school a more recent dispute over the ideal AI language concerns the difference between functional, logic and object-oriented languages. The computational ingredient of functional languages is function composition while for logic languages it is deduction. Object-oriented languages are more intuitive than formally based, having been abstracted from the requirements of simulation. (Another dispute is the question if object-oriented programming is declarative; this can be seen as an echo of the declarative versus procedural argument.) The precise nature of an object in object-oriented programming is not altogether clear with different proponents emphasizing different features. A monitor (Brinch Hansen, 1973) (a device for controlling process access to data) can even be thought of as an object. A minimalist object-oriented system would just be a set of

asynchronous communicating processes as in Hewitt's (1985) Open Systems. In the jargon of object-oriented programming, an object definition, the process code, is called a class. Classes specify prototypical entities in terms of possible operations (actions) and additionally encapsulate data structures and initialization conditions. An object is an instance of a class: a process invocation. Some object-oriented languages, such as Smalltalk (Goldberg and Robson, 1983), have concentrated on the programming environment aspects and others such as Actors (Agha, 1986) have been concerned with concurrent computation in distributed systems.

7 Reactive systems: a critical programming task

The Handbook of Artificial Intelligence (Barr and Feigenbaum, 1982, section VI.C) gives four language features that are deemed to be particularly important for AI applications programming. The first is the ability to define data structures which are expressive and easy to handle. The second is the existence of flexible control structures, in particular coroutining. (In von Neumann machines, parallelism is simulated by coroutining. Coroutines are similar to subroutines except that there is no subordination of control. That is, transfer is not hierarchical, coroutines are symmetric as regard the transfer of control.) The third is pattern matching for handling data and choosing procedures to execute. Pattern matching provides procedure selection, parameter passing and access to the components of compound data structures all in one and the same high-level mechanism. The fourth is an enhanced programming environment because artificial intelligence programs tend to be large and in a constant state of flux. In interactive systems, dialogue between the programmer and the machine occurs within a programming environment incorporating some form of editor, which can be very sophisticated. The development environment isolates the user from the intricacies of the host computer so that she can concentrate on the problem at hand. Programs may be rapidly developed, tried and easily modified and corrected in a highly user-friendly manner. This flexibility gives the programmer a large amount of responsibility in the quality of the resulting software and with low-level languages it is all too easy to lapse into idiosyncratic and bad programming practice. Interactive program enhancement, subsequent to the development of an initial attempt is rightly criticized by software professionals in the numeric domain where algorithms are well defined, but is not so easily justified in AI applications. With structured compiled languages, like Pascal, the programmer usually does not have the benefit of a sympathetic environment and the responsibility for good programming practice is shifted to the language. For this last language ideal proposed by Barr and Feigenbaum it is not necessary that the environment and the language be related but there are advantages if an environment can be programmed in the language itself. This allows programmers the ability to develop interfaces and new languages which are specific to problem domains. This gives an added flexibility to what now becomes an implementation language. Declarative languages (with the exception of object-oriented) tend to be ill-suited for such programming tasks and extensions of the kind required tend to compromise the ideology of the languages.

The choice of language is a self-imposed restriction on thought and problem solving. A language-oriented computer (or microprocessor) inherits the features and limitations of the language it supports, so these have to be considered carefully. For any machine to be generally usable it needs an operating system. Systems programming software, the generic term for operating system cum environment programs, has certain characteristics that distinguish it from other kinds:

Firstly, operating systems treat programs as if they were data. Other examples of such programs include programming environments, compilers and debuggers. An analogy can be drawn between metaknowledge, which treats knowledge as data, and systems programs which treat other programs as data. Meta-X can be understood as "X about X" so that metaknowledge is knowledge about knowledge, metareasoning is reasoning about reasoning and a metaprogram is a program about other programs. This suggests that a metalevel approach may provide a convenient connection between AI and systems programming.

Secondly, two types of programs can be distinguished, transformational and reactive (Harel and Pnueli, 1985). Transformational systems begin with the input of data, transform the input, output

the result and terminate. This is just how an unintelligent machine would behave. It would input data wholesale, process it algorithmically and output it. On the other hand, reactive programs continuously interact with the environment. Their reason for being is not the output they produce on termination, but rather, their continuing interaction with the environment; sometimes reactive programs are effectively perpetual (they do not terminate). An intelligent machine would acquire knowledge as it needed it, infer from it and output reasonings as external situations demanded. Systems programs such as environments with multiple windowing systems, operating systems and event driven real-time systems are more conventional forms of reactive programs. Reactive systems consist of conceptually parallel processes that communicate with each other and cooperatively respond to events that occur indeterminately in real-time. The resource management processes which make up an operating system cooperate to provide a virtual machine to a user. There is clearly another connection here between reactive systems and AI via Hewitt's Open System computational model, Actors (Agha, 1986). A symmetric view of input/output can be taken if the environment in the form of hardware (screen, keyboard etc.) is treated on the same footing as the software, namely as reactive components. The whole system of hardware and software is then just a collection of reactive agents some soft some hard. The natural means of synchronization of the reactive agents is condition synchronization since this is how hardware units communicate. With condition synchronisation, components suspend execution until some condition regarding their input or output has been met.

Detailed analysis of purely transformational programs which only advocate parallelism to improve performance generally reveal reactive components which maintain continuous interaction with each other during the lifetime of the computation. To fully exploit a parallel machine using a high-level programming language requires that the language can express the creation, interconnection, synchronization and communication of reactive component processes.

Earliest experience of reactive systems was gained from operating systems. Operating systems are particularly difficult to write and the advantages of using high-level languages for programming them has already proved itself with languages such as "C". (Note that "C" has to resort to operating system calls in order to describe concurrent behaviour.) High-level systems programming languages lead to significant reductions in coding effort, code size and bugs. They also provide portability and relative ease of modification. To support systems programming, a language should generally be capable of representing processes, communication, synchronization, dynamic process creation and termination, indeterminate actions, computation and resource control. Languages with these capabilities are generally called concurrent languages. Ideally then, for systems programming, the language chosen for the middle way should be naturally capable of describing reactive systems. In general, declarative languages tend to be ill-suited to such a task.

Functional languages are ideologically transformational. They represent time independent functions; notions of cooperation and synchronization are generally alien. In particular, functional languages are ill-suited to handle indeterminism which characterize reactive systems. In non-determinate systems, all branches at a choice point are fully investigated. This in general gives a number of possible solutions and of course the attendant combinatorial explosion. In indeterminacy, only one branch of a choice point is pursued; this will give rise to at most, one possible solution. In this sense Prolog is non-determinate. Functional languages are by nature determinate, there are no choice points. A variety of suggestions have been put forward to overcome this difficulty with functional languages such as primitive merge operators. However, these mechanisms undermine the ideology which is the *raison d'etre* of the languages. Because logic languages are relational they do not have the problem of accounting for indeterminism, it is already built in because the structure is relational and not functional. In general, however, the indeterminism is manifested as non-determinism: large search spaces that may counteract any gains of parallel processing. For instance, in an operating system only one solution as to how a file can be written to disc is required. The objects of object-oriented languages by their very nature, are reactive agents and so these languages tend to be well suited to systems programming.

High-level languages deliberately impose structure on the problem solving behaviour of the user

in order to make certain problem expression more natural and to discourage bad programming practice. They encapsulate high level problem solving concepts (such as recursion) that generally make reasoning about programs simpler. The drawback is that restrictions on the allowed data structures, control structures and primitives make some algorithms more difficult to express than others. It is not particularly useful to measure the expressive power of a language in terms of the class of functions (or relations) it can evaluate, since all programming languages tend to be Turing complete. If the expressive power of a language were to be measured by the variety of algorithms (or paradigms) it is possible to express naturally, (the word naturally causes some problems in definition) then, higher level languages would generally be less expressive. A good example of this is simulation which attempts to model real life situations in which several activities occur concurrently and interact with one another. In this guise parallelism is not used to promote efficiency; rather it is being used because it leads to a more natural representation of the problem. Writing a solution as a concurrent program may be the best approach even when execution of the program is carried out on a uniprocessor machine, as in coroutining. From the previous discussion, an equally important measure of expressiveness is how easy is it to write an operating system or a windowing system in the language.

8 Specification, logic and Prolog

With the increased sophistication demanded from software, in particular reactive systems, it is increasingly complex and all too often bug-ridden. In theory many approaches have been proposed for debugging and increasing confidence in the correctness of a program, some dealing with programming methodology and others with algebraic proofs that a program does what the specification intends of it. In practice, for large programs, the most widely used methods are comprehensive test data and tracing of the computation. The art of bug detection with profilers, mutations, bounty hunters and black teams has become a jargon generator in its own right. Increasingly, software developers begin with a specification of the functionality required by the software (particularly those developers with penalty clauses in their contract). Automatic verified code generation from a specification appears to be an emerging trend with the use of Fourth Generation languages and sales of programmer workbenches gaining ground.

Insight into the relationship between specifications and programs can be drawn from some definitions proposed by Hoare (1982) with respect to the concurrent programming language CSP. "A specification is a predicate describing all observations of a complex system." The word predicate could be generalized to a set of logical formulas or sentences but Hoare goes on to effectively include this possibility. "Specifications of complex systems can be constructed from specifications of their components by connectives in the Predicate Calculus." Some would argue that the limitation to Predicate Calculus should be relaxed to encompass other more general forms of logic, for example modal logics. However, for the purposes of efficient mechanical theorem proving only Horn Clause Calculus has, as yet, proved tractable.

Continuing from the previous definitions, Hoare defines a program: "A program is just a predicate expressed using a restricted subset of connectives, codified in a programming language." If "a restrictive subset of connectives" is replaced by a restricted subset of Predicate Calculus, namely Horn Clause Logic, then the specification is a logic program and no coding is necessary.

The declarative reading of a logic program gives partial information about the run-time behaviour of the program, it bounds the possible answer set. Semantics based on logical consequence is much more natural than other styles of programming language semantics. In particular, the meaning of a logic program does not of necessity require the construction of models using category theory as do those based on the work of Scott and Strachey or those in the abstract-data-types literature based on initial and final algebras. The consequence of a logic program are all those theorems which must be true whenever the premises which constitute the program hold. This is one more of the reasons why logic languages are known as very high-level and why they have been chosen by the Japanese as the foundation of their Fifth Generation initiative.

It is now becoming recognized that Prolog is a *good thing* for database applications, problem solving, expert systems, natural language parsing and compiler writing. However, the present fashion for logic programming in response to the Japanese Fifth Generation initiative has been widely criticized. Typical of the criticism is the following taken from *The Guardian* (23 May, 1985) and attributed to Winograd: "There is a small class of problems for which Prolog works great and you can do beautiful demonstrations on these problems. But when you have to deal with time-dependent behaviour like an operating system, you get away from the nice qualities of Prolog, you have to use the ugly features and you lose the advantage. For commercial users of a language it is such large operating system-type software which they most need to program. Will they want an interpreted language which can't usefully be used in many problems?"

9 Criticisms of Prolog

By the late 1960s, before logic became fashionable, the language that came closest to Barr and Feigenbaum's ideal AI programming language was LISP, a functionally (lambda calculus) inspired language, it was used for the majority of applications. LISP was a list processing language but it was certainly not the first attempt at a list processing language. The language developed by Newell, Shaw and Simon (1956) for the logic theorist, called IPL was a list processing language which influenced the development of LISP (McCarthy, 1978); LISP, however, was much more elegant. LISP provided powerful and flexible facilities for manipulating symbolic information but at the time appeared to give little direct support for implementing control and data structures. This led to a number of proposals more tailored to the needs of AI. One of the foremost of these was Planner (Hewitt, 1969). Planner incorporates automatic backtrack control, pattern directed database search, and pattern directed invocation of procedures. The similarities with Prolog are easily apparent and according to Dowson (1984) the development of Prolog was influenced by the implementation of MicroPlanner a subset of Planner. By 1972, when the theorem prover on which Prolog is based was proposed (Kuehner, 1972), the initial euphoria associated with Planner had evaporated and some of the basic deficiencies had become apparent. Planner was effectively displaced by Conniver (Sussman and McDermott, 1972). The criticisms of Planner, by Sussman and McDermott (1972) apply almost word for word today, replacing Planner by Prolog.

The criticisms of Planner (Sussman and McDermott, 1972), with regard to its naive search strategy apply both to Prolog and its proposed OR-parallel extensions (Warren, 1987). The OR-parallel extensions of Prolog advocate replacing the backtracking search for alternative solutions by parallel search. As noted previously the problem with OR-parallelism is that when it is used recursively at each goal in the search tree, the generated parallel activities grow exponentially with respect to the number of levels in the search tree. Current OR-parallel implementations resort to backtracking when all the processors are saturated. Problematic for Prolog is that using the most general unifier the alternative solutions are not independent as was previously suggested because of the logic variable. The logic variable acts as a template which is filled in as the computation proceeds but OR-parallelism requires that the branches at a choice point have the same initial template; a simple though computationally expensive solution requires the duplication of the template for each OR-parallel branch. Current research in OR-parallel Prolog proposes more sophisticated schemes which reduce the overhead (Warren, 1987). In particular the nondeterminism is treated lazily rather than eagerly.

A common use of backtracking is *generate and test*. It is a long established approach (cf. Erastosthenes Sieve for determining prime numbers) for problems where there is a need to filter out relevant data items from a mass of irrelevancies. Prolog provides a compact notation for encoding this search, but initiates find it difficult to understand program behaviour because the flow of control is not explicit. The problem is that the syntax of a Prolog program does not reveal that there is one gigantic nest of failure driven loops in any query, and every subgoal that eventually fails is only a tiny part of it.

The anonymity of the clauses that may be invoked by Prolog's head/call unification mechanism

pretends that there are many "independent" methods of solving the same problem. Not only does this organization force each clause to be used in complete ignorance of clauses that have been tried so far but also the others that have not been tried. In many cases different clauses will either duplicate solutions or the same unacceptable answer fails or runs out of space for the same reason. Novice Prolog programmers find the duplication of solutions annoying and difficult to understand. Exhaustive search is only appropriate when there is insufficient metaknowledge about the problem in hand. Prolog programmers use metaknowledge to prune the search tree in the form of clause ordering and the infamous cut. This metaknowledge is used implicitly despite the claimed virtue of explicit declaration by Prolog enthusiasts. It is the liberal use of cut which most critics of Prolog mainly cite in their arguments against it.

A program cannot be described as "intelligent" if it does not make intelligent decisions. Search and deduction are core intellectual skills or as Newell (1981) put it: "All intelligent activity is based on search". An intelligent person does not make an exhaustive search in any prior order but is guided by what is found. It is necessary to examine the assumptions leading to failure and to learn from mistakes. In Prolog, alternative program control can be achieved by writing metacircular interpreters (interpreters for Prolog written in Prolog) which are enhanced to generate lemmas and perform "intelligent" backtracking (Codognet et al, 1988). The uniformity of clauses (the program) and data (the terms) make Prolog, with its pattern matching ability, highly suitable for such metaprogramming. Languages that can represent programs as data make it natural to bootstrap new languages or control by metaprogramming. This makes it possible to solve complex problems not by writing complex programs but by designing flexible problem or user oriented programming languages and implementing them with a minimal amount of effort. This is how both Prolog and LISP accommodate the second desirable property of Barr and Feigenbaum: flexible control structures.

Meta-interpreters, however, run slower than direct execution of the object program, so they achieve flexibility at the expense of efficiency. This drawback can be overcome to some extent by the fashionable transformation technique of partial evaluation (New Generation Computing, 1988) which combines a meta-interpreter and an object program into a more efficient but specialized program. The transformation of search intensive programs to search free programs using metacontrol knowledge is another example of partial evaluation. Partial evaluation provides a common framework for discussing program transformation, program control, programming language interpreters, compilers and a means for both compiling and for automatically constructing compilers from interpreters. The result of partial evaluation of an inference engine with respect to the inference rules is a specialized inference engine just for those rules. This removes some of the inefficiency of meta-interpretation for designing problem solvers and application specific programming languages.

Negation as failure is often cited as an attractive feature of Prolog. Horn Clauses can only infer positive information by deduction. The closed-world inductive hypothesis is that the information about the world being modelled is complete in the sense that the relations that hold are exactly and only those that are deducible from the axioms. Although Prolog is not complete it is well suited to the closed-world assumption and negation as failure (those goals which fail are a subset of the complement of those that can be proved) generally works reasonably well and enhances the expressive power of the language. At first glance it might seem that the closed-world assumption is intelligent because it provides a ready made default for any goal. Unfortunately the default answers become less and less realistic when incompleteness is introduced either accidentally because of infinite branches of the search space or deliberately by the use of cut. This is compounded in open systems where one expects continuous growth of knowledge with attendant inconsistency among knowledge bases.

What is gained by chronological backtracking is clear and appealing. In the first place it provides a mechanism for generating alternatives one at a time, to be used in an effort to prove some goal. Secondly, it provides a mechanism for eradicating the consequences of alternatives later found to fail and thus economizes on space usage. And thirdly, it simulates induction by deduction in the form of negation by failure. Its basic defect is that it forces an expensive assumption that the altern-

atives at each decision point are independent; that the failure of one of them will produce no information which can influence the selection of further alternatives. Sussman and McDermott (1972) claim that backtracking (the same argument can be applied to OR-parallelism) as a means of search are of questionable usefulness for applications in AI. Programs which use exhaustive search are often the worst algorithms for solving a problem. The ubiquity of implicit search and the illusion of power that a query gives the user merely by use of invisible failure-driven loops encourage superficial analysis and poor programming practice. It is this approach to AI that gave it a bad name and led to Lighthill's (1972) pejorative use of *combinatorial explosion* which set the UK's research in AI back at least ten years. A possible solution is to abandon the myth that there could be several independent methods of attacking any interesting problem and concentrate on techniques for making searches in different branches relate and cooperate with one another. In the context of parallelism this means using AND-parallelism or concurrent processes in preference to OR-parallelism. Again there is empathy here with Hewitt's Open Systems.

10 Concurrent logic languages

Previous discussion on knowledge representation and program specification indicated that Prolog might be an ideal candidate as the language for the middle layer. The above remarks on its unsuitability for reactive programming and the criticism of the implicit search has cast doubt on this idea. In addition to its logic based semantics, the great redeeming feature of Prolog is its affinity for meta-interpretation and partial evaluation. Previous discussion on the nature of AI has recognized the importance of the metalevel, and the analogy with systems programs indicates how it might provide a natural platform for reactive system programming. Meta-interpretation and partial evaluation are powerful and seductive ideas on which to build an AI machine (or microprocessor). A problem, though is to build a machine (processor) which will perform well enough to make meta-interpretation a viable proposition. The prospect of greater performance is one but not the only desirable aspect of concurrency. Concurrency may be used to provide fault tolerance and an extensible computer system where the performance may be enhanced by adding more processors, rather than replacing the whole system by the latest faster model. On the software side, concurrency provides the ability to directly model real life problems in which several objects interact with one another. Previous discussion has noted that the computational model of Actors, an object-oriented language proposed for distributed AI problems, provides the sort of expressive power desirable for programming reactive systems. A process based computational model for Horn Clauses (the theoretical foundation of Prolog) rather than Prolog's subroutine model would then seem eminently desirable.

Designers of Prolog implementations have been driven by the criticisms of the slowness of execution. Great achievements in performance have been made since the first implementations but these have been attained at the expense of further specialization to the von Neumann architecture. Prolog has a single thread of control necessarily associated with block structured languages and stack manipulation. The single threading limits the range of applications that Prolog programmers can build, for example systems which involve multiple cooperating processes such as in reactive programs are extremely difficult and inefficient to program in Prolog, even using meta-interpretation.

A Prolog meta-interpreter for an operating system (if one were possible) could simulate concurrency by coroutining. However, the underlying depth-first search strategy cannot gurantee any fairness in the scheduling of coroutines. Many AI algorithms are indeterminate in nature, so dynamic resource allocation and load balancing are of major concern in parallel processing. The strength of a high-level language based environment is that it allows the programmer to interact with the underlying machine solely in terms of the computational model of the language. Any such language must be able to represent all aspects of the operational behaviour of the machine: dynamic process creation, concurrency, communication, synchronization, process termination, indeterminate actions, computation control and resource control. Prolog with depth-first search and backtracking is a natural consequence of the stack implementation and as a result shows the influence of the von Neumann architecture. Essentially, Prolog was designed for efficient evaluation on von Neumann archi-

tectures and doesn't address the fundamental issues of concurrency. It is difficult to see how a language based on a fixed number of stacks can achieve the desired result of a machine in which a lot more silicon is active at any one time than is usual in a von Neumann architecture. By the very nature of a stack, only the top is generally active.

Closer examination of the Prolog computational model reveals that it is not the only possible execution pattern that can be founded on Horn Clauses. For Prolog, the exploitation of parallelism is complicated by the non-directionality of the modes of variables, and backtracking (Ringwood, 1988a). The computational model of Prolog is stack based because this is most suitable for implementation on a von Neumann machine. Depth-first search and chronological backtracking is a natural partner of a stack based implementation. In a resolution step the selected literal is the top of the goal stack and the newly introduced literals from the body of a unifying input clause are pushed on top of the stack. This gives rise to the hierarchical subroutine computational model of Prolog in analogy with block structured languages such as Pascal. This is not the only possible computation model that can be founded on Horn Clauses: an alternative model suitable for medium grain parallelism can be realized by means of a goal queue. The selected literal is the head of the queue of literals; new literals from an input parent are added to the end of the queue. This gives rise to a process interpretation (or even object-oriented interpretation) of goal literals, and this observation, that goals can be treated as processes (objects), is the essence of Concurrent Logic Languages. Synchronization is re-established in this model by condition synchronization (which was deemed to be desirable for reactive systems) as an extension of pattern matching. The model has the capacity for fair scheduling, does not embody exhaustive search (in view of the criticisms of backtracking and OR-parallelism, exhaustive search was an expendable feature) yet retains the desirable feature of affinity for metaprogramming and partial evaluation. Languages based on this model called concurrent logic programming languages (CLPs) are not to be confused with constraint logic programming languages. For two descriptions of how these languages can be derived from Prolog see Ringwood (1987, 1988a).

There are now a large number of languages using the process execution model of Horn clauses and the family is still growing. With a few exceptions this output has been generated by three competing research teams: ICOT (Institute for New Generation Computer Technology), Japan; Imperial College, UK and the Weizmann Institute, Israel. Fortunately, the seemingly exponential growth in the number of languages is only an illusion. There are only a finite number of features that come out of comparing these languages (Ringwood, 1988c) and the number of languages that can be formed by composing these features is finite. (The article (Ringwood, 1988c) is a follow-up to the present article and gives more detail on the form of concurrent logic languages for the interested reader.)

Concurrent logic languages, naturally enough, bear a close resemblance to other medium grain process based real languages such as Occam (Inmos, 1984) and computational models such as CSP (Hoare, 1985), CCS (Milner, 1980) and Actors (Agha, 1986). The main difference between CLP and the others is that communication is by shared variables rather than message passing. Despite this fact, for CLPs the communication mechanism does bear many of the hall marks of message passing systems and comparison can be made in these terms. For CSP and Occam communication is synchronous while for CLPs and CCS it is asynchronous. Actor objects and CSP processes address each other by name but for CLPs and Occam, communication is via named channels. Communication via channels has the advantages of modularity and abstraction. Channels are more abstract since knowing a channel conveys less information and is more flexible than named cooperating processes. This is one of the departures of Occam from CSP brought about by practical considerations.

11 Conclusion

Far from sinking into respectability, the art of logic and its mechanization has been turbulent, intense and schizmatic. The Japanese proposal for Fifth Generation machines based on logic programming has been much criticized but is not as radical as has been suggested. It is just following a

well established engineering practice, namely breaking a task into layers, and building on well understood foundations, such as logic. The programming language Prolog, provided a starting point as the middle layer for representing and processing knowledge from which applications and hardware can be developed. The feedback loops of design from the applications, particularly reactive systems and efficiency considerations have led to shifts in the kernel language to concurrent logic languages. This has also been accompanied by a change in attitude to the role of Horn Clause Logic, the theoretical foundations of logic programming. In this new interpretation the place of Horn Clause Logic is not at the object level but rather at the metalevel. This is due to the fact that affinity for meta-interpretation was one of the indispensable features of Prolog but completeness of the theorem prover was dispensable.

It is still an act of faith that the challenges of designing an AI machine can be met by building on the foundations of logic, software engineering, computer architecture, networking and VLSI technology. A further act of faith is that metalevel programming can provide the necessary layers to produce a better match between the parallel hardware and AI applications. Neural machines are beginning to challenge the Fifth Generation before it has even got off the ground but this need not necessarily be seen as a challenge if it is agreed that symbolic abstractions and neural nets are considered as different complementary levels of understanding in much the same way as conscious and unconscious thought processes take place in the human brain.

A rather greater threat to the Fifth Generation initiative than neural machines is the rate of development of general-purpose machines. For example, LISP now runs faster on RISC machines than it does on outdated special purpose machines such as Xerox workstations. In the author's opinion a safer prospect than the Japanese approach of designing special purpose hardware is to adapt concurrent logic languages to commercially available, upward compatible, RISC microprocessors which give hardware support for process based languages, such as the Transputer. Occam, the kernel language of the transputer, is minimal and efficient because of hardware support. However, the language is inflexible, does not support recursion, abstract data types, dynamic data structures and dynamic process structures. Occam code is not a first class object and so the language is not well suited to metaprogramming and partial evaluation. Concurrent logic languages could provide the ideal high-level language to program these processors for both AI applications and reactive systems.

Acknowledgements

The author is grateful to Matthew Huntbach for pointing out the analogy between on the one hand, the development of Conniver from Planner and on the other, the development of concurrent logic languages from Prolog. The author is also grateful to Alastair Burt for useful comments on a previous version of this article, to John Fox, editor of *The Knowledge Engineering Review*, for insisting on changes which have brought the article to its present condition and Bob Kemp and Sasikumar M for proof reading the current version.

References

Agha, G, (1986). *Actors: a model of concurrent computation in distributed systems* MIT Press.

Barr, A and Feigenbaum, E A, (1982). *The Handbook of Artificial Intelligence 2* Pitman.

Brinch Hansen, P, (1973). *Operating System Principles* Prentice Hall.

Chang, C L, (1970). "The unit proof and input proof in theorem proving" *JACM* 17 698–707.

Codognet, C, Codognet, P and File, G, (1988). "Yet another intelligent backtracking method" *Proc. Fifth Int. Conf. and Symp. on Logic Programming* Kowalski R and Bowen K (Eds.) MIT Press.

Colmerauer, A, (1973), "Les systems-Q ou un formalisme pour analyser et synthetiser des phrases sur ordinateur" *Publication Interne No 43*, Department d'Informatique, Universite de Montreal.

Colmerauer, A, (1982). "Prolog and infinite trees" In: *Logic Programming* Clark KL and Tarnlund SA, Academic Press.

Devlin, K, (1987). "A clever little number" *The Guardian* July 16th.

Dowson, M, (1984). "A note on Micro-Planner" In: *Implementations of Prolog* J A Campbell (Ed.) Ellis Horwood.

Elcock, E W, (1988). "Absys: the first logic programming language—a retrospective and commentary" to be published *Jnl Logic Programming*.

Gardner, M, (1982). *Logic Machines and Diagrams* Harvester Press.

Goldberg, A and Robson, D, (1983). *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.

Green, C, (1969). *The Application of Theorem Proving to Question-Answering Systems* PhD thesis, Stanford.

Harel, A and Pnueli, A, (1985). "On the Development of Reactive Systems" In: *Logics and models of Concurrent Systems* Apt K R (Ed.), Springer Verlag.

Herbrand, J, (1930). "Reserche sur la theorie de la demonstration" These, U. de Paris. In: *Escrit logiques de Jaccques Herbrand PUF*, Paris (1969).

Hewitt, C, (1969). "*Planner: a language for proving theorems in robots*" *Proc IJCAI* **1** 295–301.

Hewitt, C, (1985). "The challenge of Open Systems" *BYTE* April 1985, 223–42.

Hoare, C A R, (1982). "Specifications, programs and implementations" Oxford University *Tech Monograph PRG-29*, Oxford.

Hoare, C A R, (1985). *Communicating Sequential Processes* Prentice Hall.

Inmos Ltd, (1984). *Occam Programming Manual* Prentice Hall.

Jaffar, J and Lassez, J-L, (1987). "Constraint logic programming" *Proc 14th ACM POPL Conference*, Munich.

Kuehner, D, (1972). "Some special purpose resolution systems *Machine Intelligence* **7** 117–128, Edinburgh University Press.

Lighthill, J, (1972). Artificial Intelligence Reports to SERC, HMSO.

Loveland, D W, (1969). "A simplified format for the model elimination theorem-proving procedure" *JACM* **16** 349–63.

Loveland, D W, (1970). "A linear format for resolution" *Proc of the INRIA Symposium on Automatic Demonstration* **1968** 147–162, Roquenfort, Springer-Verlag.

McCarthy, J, (1958). "Programs with common sense" *Proceedings on the Symposium on the Mechanization of Thought Processes* National Physics Lab, Teddington, England.

McCarthy, J, (1978). *History of LISP, ACM SIGPLAN Notices* **13** (8) 5, 11.

Milner, R, (1980). *A Calculus of Communicating Systems* LNCS **92,** Springer Verlag.

Newell A. "Duncker on thinking: an enquiry into progress in cognition" Carnegie-Mellon University, Technical Report CS-80-151.

Newell A, Shaw J C and Simon, H A, (1956). "The logic theory machine: a complex information processing system" *IRE Trans on Information Theory* **2** 61–79.

New Generation Computing, (1988). Selected Papers from the Workshop on Partial Evaluation and Mixed Computation **6** (2,3).

Ringwood, G A, (1987). "Pattern-directed, Markovian, linear guarded, definite clause resolution, submitted *JLP* but rejected after the unreasonably long time of 21 months.

Ringwood, G A, (1988a). "Parlog 86 and the dining logicians" *CACM* **31** 10–25.

Ringwood, G A, (1988b). "SLD: a folk acronym?" *Logic Programming Newsletter* **2/1** 5–8.

Ringwood, G A, (1988c). "A comparative exploration of concurrent logic languages" submitted *The Knowledge Engineering Review*.

Ringwood, G A, (1989). "SLD: a folk acronym?" To be published *SIGPLAN Notices*.

Robinson, J A, (1965). "A machine-oriented logic based on the resolution principle" *JACM* **12** 23–41.

Schofield, J, (1988). "The taxonomy of computer evolution" *The Guardian* 29 Dec 1.

Sussmann, G J and McDermott, D V, (1972). "From planner to conniver—a genetic approach" *Proc AFIPS Fall Conference* 1171–1179.

Ungar, D and Patterson, D, (1987). "What price Smalltalk" *IEEE Computer* **20** 67–72.

Warren, D H D, (1987). "OR-parallel execution models of Prolog" In: *Tapsoft 87*, LNCS 250, Springer-Verlag.