REAL-OPS
A Real-Time Engineering Applications Language For Writing Expert Systems*

W. B. Dress
Instrumentation and Controls Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831

Abstract

This work describes how Forth was used as a descriptive language for rewriting the expert-systems language OPS5. The goal was to produce a multitasking applications language for real-world, intelligent control problems. The next logical step--a derivative REAL-OPS running on a high-speed Forth engine--is discussed. The resulting integration of Forth, Forth engines, and expert-system technology in the mid-1980s will provide the high performance needed in the time-critical expert systems required for intelligent control of military and industrial systems now planned for the mid-1990s.

Introduction

As the success of artificial intelligence applications became evident in the area of expert systems for medical diagnostics[1] and computer configuration[2] problems, it was just a matter of time until extending the methodology to problems of real-time process control and data reduction was attempted. First attempts showed clearly that execution speed would be a limiting factor for any real-world control problem, so attention was given to making LISP machines run faster and providing I/O channels with higher bandwidth. One of the successes with a complex expert system digesting large amounts of incoming data and providing expert decisions in real time[3] showed how effective such an approach could be.

However, the problems of efficient access to the inference engine and working data set by asynchronous external events were largely ignored, probably being left for hardware manufacturers to solve via the "bigger and faster" route. This seems an unsatisfactory state of affairs in that only those institutions able to afford the high-end, newly developed LISP machines are able to consider applying real-time expert systems to their problems.

*Research performed at Oak Ridge National Laboratory, operated by Martin Marietta Energy Systems, Inc., for the U.S. Department of Energy under Contract No. DE-AC05-84OR21400.

REAL-OPS is an attempt to fill the gap between merely running a system faster (waiting until the necessary data are present at an I/O port) and the asynchronous, multitasking operation of a real-time, event-driven system. The idea is to start with an effective real-time, multitasking software base and build into it the necessary expert system capabilities in a manner recognizable to both the expert system and real-time control communities. To this end, a multitasking version of Forth[4] containing the necessary interface words to the IEEE-488 instrument bus was chosen for the Hewlett-Packard Series 200 and 300 desktop computers. Thus the problems real-time, multitasking, and access to external events were neatly solved, leaving the problem of integrating an established expert-system language with an underlying Forth base.

The expert systems language chosen for embedding in Forth was OPS5, a widely used production-rule language. The reasons for this choice were twofold: first, OPS5 is a powerful, forward-chaining, efficient language; second, it is conceptually and syntactically simple. The first property gives the means to produce a language for engineering control applications since such problems are usually event-driven; that is, the expert controller must respond quickly and correctly to the instantaneous data stream that conveys the current state of the system being controlled. The second property allows easy extensions and modifications to the language, producing a more powerful result tailored to the needs of real-time, multitasking expert systems.

What is OPS5?

Cosmetically, OPS5 is a scheme for constructing production rules (modules) in the form of **IF...THEN...** statements, where the **IF** part, or left-hand side (LHS), specifies a set of data patterns which must be consistently matched by a portion of the actual data set residing in "working memory;" and the **THEN** part, or right-hand side (RHS), indicates the set of actions to be carried out when the LHS is satisfied. Typically, the RHS actions make or remove data elements in working memory, initiate I/O with the operator or files, and perform necessary calculations via external calls. OPS5, developed at Carnegie-Mellon University[5] in the late 1970s, is one of the most popular and widely used production-rule languages, and the only one with a textbook[6] devoted to it. Typically, OPS5 is used for writing expert systems; but, being conceptually and syntactically simple, it has much broader applications. The Brownston book[6] devotes careful attention to deciding when an algorithmic approach should be used for a problem and when a production-system approach is more appropriate. Structure and complexity are the guide: unstructured, complex problems are more amenable to solution via a production paradigm whereas well structured, simple problems are better handled by specific algorithms. That is not to say an expert-system implementation cannot represent a well-structured problem. The often-cited "animals" expert system is usually expressed as a set of backward-chaining rules that form a static, almost algorithmic, decision-tree type system. A truly unstructured problem involving complex, dynamic data patterns would be difficult to represent in such a fashion.

Each OPS5 rule is an independent module, loosely coupled to other rules by the data set in working memory. There are no global variables in standard OPS5; all variables refer to values of working memory elements, and the particular binding is valid only within the rule where it appears. Conceptually, the set of rules in an OPS5 system may be thought of as "peering" into working memory in parallel, each one "looking" for a

particular set of data patterns. When a rule finds a set of data patterns matching its own pattern prescriptions (condition elements), it is free to "fire."

Internally, OPS5 is much more complicated than a set of **IF...THEN...** procedures. To see that this must be so, consider an expert system with one thousand rules, each rule having ten conditions and each condition consisting of a ten-element pattern. The brute-force method would be to consider each of the hundred thousand possible pattern elements as a candidate for each data instance in working memory during each system cycle, and then decide which rule to fire. Since there may be several thousand working memory elements, each with perhaps ten terms, even an efficient Forth system would take too long to check all of the several billion possible matches to make a real-time expert system feasible. The Rete algorithm[7] is responsible for OPS5's efficiency. This algorithm maintains lists of pointers from those actions potentially making working memory elements that <u>could</u> match particular condition elements to those same condition elements. The current state of the system is maintained, and only differences from the current state are noted during each pass through the "recognize-act" cycle. This differential method obviates the need for matching each data-element term against each condition-element term during each system cycle.

Rules, Condition Elements, and Actions. A rule in OPS5 consists of a collection of patterns, or condition elements, specifying particular instances of a class object being considered, followed by a list of actions to be effected if working memory elements consistently satisfying the patterns are present. Working memory is simply a large memory pool or common area where instances of data patterns are kept as long as needed. The class object, or "literalized" object as it is usually called, is merely the name of an n-tuple of attribute-value pairs, or "terms." As an example, consider a generalized "vector" which has been given the class name of **GOAL**. Each term of this vector has a name--its attribute. Thus a condition element such as **(GOAL ^STATUS READY ^ID 101)** has two terms specified by the attribute **STATUS** and **ID** (^ is a symbol indicating an attribute similar to the way an element of a record is selected in Pascal). These attributes, or slots in the n-tuple named **GOAL**, have values **READY** and **101** respectively. The condition element specifies a pattern to be matched. Working memory contains the actual patterns made by the system's rules, the operator, and, in the case of REAL-OPS, by external events via event-handling tasks. Thus a working memory element such as **(GOAL ^STATUS PENDING ^ID 101)** would not match the above condition element, whereas the working memory element **(GOAL ^STATUS READY ^ID 101 ^SUBGOAL 11)** would match. A rule consisting of just that condition element for its LHS would be able to fire. The difference between memory and condition elements is that the former are instantiated pieces of data, while the latter are potential patterns to be matched; that is, the specific types of things the rules are "looking for."

The RHS of a rule consists of a set of actions to be carried out when the rule fires. Typical actions include **WRITE** for messages to the console, **BIND** for assigning a value to a variable, **MAKE** for making new working memory elements, **REMOVE** for removing working memory elements no longer needed, and **MODIFY**, which is a combination of **MAKE** and **REMOVE**. The actions that alter the contents of working memory are responsible for driving the production system toward its goal of solving the particular problem. In an event-driven system, **MAKE** must be altered to allow external events to enter the expert system's data set in an asynchronous manner.

Of course, in a Forth-based system, one of the actions is **FORTH**, which takes as its arguments any string of symbols acceptable to Forth's interpreter. This allows access to previously written code as well as shortcuts in data manipulation and file access. Adding graphics and other enhancements is also be done by **FORTH** actions.

Writing OPS5 in Forth

The task of writing OPS5 in Forth is organized into several "chapters" as recommended by Brodie.[8] The first few chapters deal with the necessary tools and establish a hierarchical vocabulary structure. The tools chapter contains the words used throughout the entire application for such things as bit manipulation, memory management for dynamic data structures, and list bookkeeping. Since OPS5 requires a multitude of dynamic lists (for pointers, data values, stacks, etc.) as well as the ability to

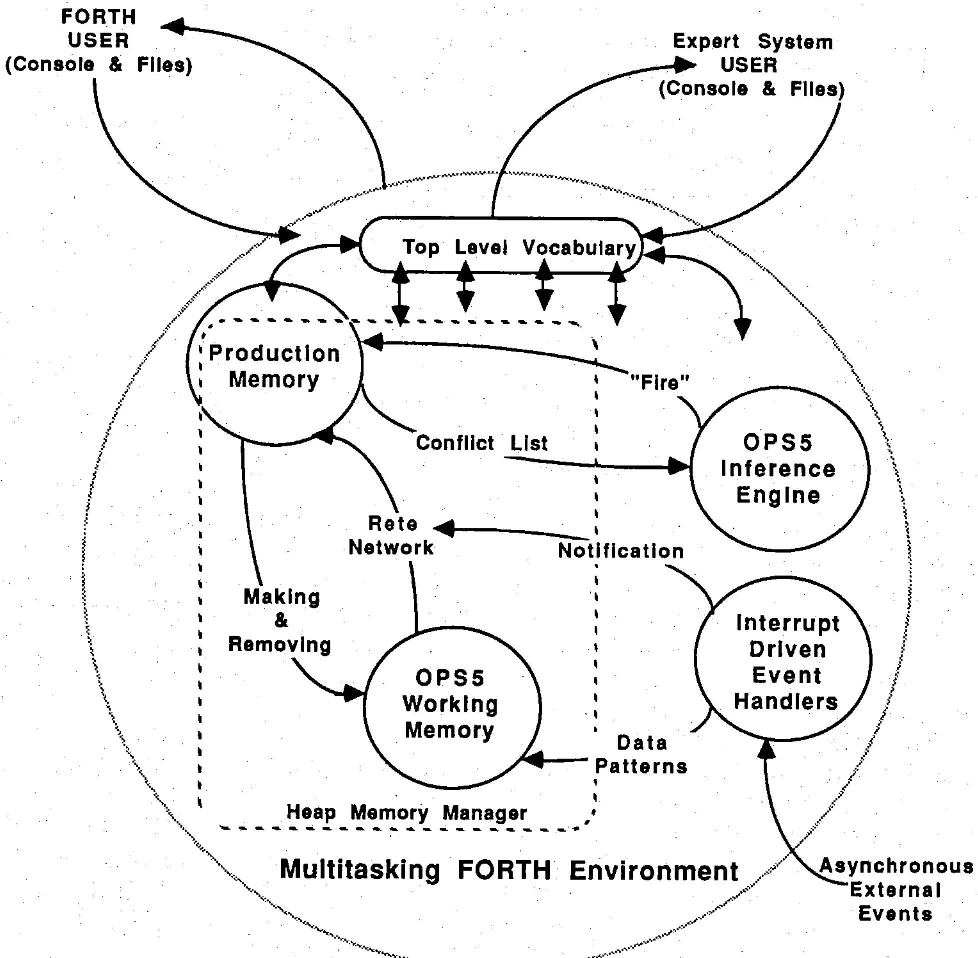

Figure 1. Conceptual diagram of REAL-OPS showing main data and control flows.

make and remove working memory elements, efficient memory management is essential to Forth implementation. Most LISP and C implementations of OPS5 practice the standard method of garbage collection[9] by marking list nodes as unneeded and deferring memory reclamation until absolutely necessary. The entire expert system must then be put in standby mode while the memory is cleansed of unused data structures. An alternate method, adopted in this work,[10] employs a synchronous mode of garbage collection, avoiding the unpredictable, deadly delays of the conventional method.

Vocabularies are principally used as a logical convenience, mirroring the structure needed to implement the OPS5 language. For example, the word **MAKE** appears in three different vocabularies--one for compiling the **MAKE** action into an efficient data structure, one for actually carrying out the action and making a new working memory element when a rule fires, and one in the top-level vocabulary to allow the operator to **MAKE** things as well. All rules are placed into a Production Vocabulary and all literalized objects into a Class Vocabulary for convenience in categorizing for searches and user access.

Figure 1 provides a conceptual overview of OPS5 embedded in Forth. Three entities are external to the system--an expert system user, a Forth programmer and user, and any number of external asynchronous events. Since Forth provides what is usually referred to as the operating system, all communication actually takes place either via Forth's interpreter or by means of drivers written to handle external events. The memory partition common to OPS5 is shown with the major flow of control. Certain pieces are located in the heap memory partition, while other sections are found in the usual Forth vocabularies. The structures located in the heap have pointer references in the various system words, making everything accessible from Forth words.

The Parser. The next REAL-OPS chapter contains the parser, which is implemented as a transition network and makes extensive use of the **DOER–MAKE** construct[8] for vectoring various words into the parser framework. A special state stack is maintained in the heap, allowing parser states to be nested. For example, when parsing a condition element, the initial state is one expecting a class name. Once the user (or file system) responds with a valid class name, the parser state changes to one expecting a pattern term via the phrase **STATE TERM.STATE**, which switches the parser state to that expecting an attribute-value pair and pushes a reference to **TERM.STATE** onto the state stack. **TERM.STATE** in turn switches to **VALUE.STATE** upon entry of a correct attribute of the class being considered. The word **PRIOR.STATE** is executed when the pattern for the value of the slot being compiled is entered. This merely pops the state stack and executes the word pointed to by the new top stack element (switching back to **TERM.STATE** in this case). **PRIOR.STATE** is itself vectored from within a word called **END**, which is executed after each symbol is entered by the user, effectively maintaining the state stack at its correct level for parsing any legal phrase in the OPS5 syntax.

LHS and RHS Compilers. The next implementation chapter contains the mechanism for building the data structures representing production rules and compiling the condition elements into data structures accessible to the rules. Similarly, the mechanisms for building the RHS actions into the rules are collected into a separate chapter. Pointers are maintained in the heap from each class object to the condition elements of that class, similar to establishing inheritance in Smalltalk objects. The Rete network is built by matching each new action to all previous condition elements, and each new condition element to all previous actions as the rule set is being compiled. If

an action could conceivably produce a working memory element satisfying a condition element, a pointer to that condition element is added to the network list for the action.

Bit maps for the satisfied condition elements in a rule and the terms present in a condition element are maintained while the system operates. Logical comparisons of bit maps speed the matching process and conflict resolution. Each rule also contains a pointer to a list specifying the current state of working memory as viewed from that rule.

RHS Actions. The heart of the system is the chapter containing all the words for effecting the RHS actions in a running system. The simple action of making or removing a working memory element may affect every rule in the system, so a means of matching the potentialities pointed to by the Rete network is needed. If a new working memory element actually matches a rule's pattern, a consistency check needs to be made. A rule waiting for a variable **<x>** and another variable **<y>** where the value bound to **<y>** is not to be greater than the value bound to **<x>** would be matched by all numerical instances of **<x>** and **<y>**, but perhaps not consistently so (try **<x>** = 5 and **<y>** = 11).

Note that each LHS may consist of several condition elements (patterns) and that each condition element may have many working memory elements independently (or disjointly) matching it. The set of possible matches is the power set, or product of the sets of working memory elements associated with each condition element in the LHS. This can easily become a very large set; it is not uncommon for a rule to have 25 condition elements, each with as many as 50 working memory elements. The power set contains 50^{25} elements, an astronomical number indeed. It would be hopeless to attempt to examine each member of the power set for consistency, so, following Forgy,[7] partial sets are constructed and checked for partial consistency. Consistency is checked starting with the first condition element and its most recent working memory element. Then the next condition element is examined, starting with its most recent working memory element, and so on. If the last condition element is reached and shows consistency for one of its matching working memory elements, the set is consistent, and the rule is placed in the conflict set. Should any of the sets of working memory elements become exhausted before the rule can be declared consistent, the rule is not yet ready to fire and the process terminates. Pointers into the lists of matching working memory elements are kept with each rule so that the process does not need to be repeated in its entirety each time the rule needs checking. Thus the problem is far less than indicated by the size of the power set.

Conflict Resolution. The recognize-act cycle mentioned above culminates in a set of satisfied rules. Since only one rule may fire in each cycle, conflict resolution applies one of several strategies to pick the winner. Deciding on the winning rule involves sorting all rules in the conflict set (which is implemented as a list of rule pfa's) by recency of the working memory element satisfying the rule's first condition element--the rule with the most recent element wins (Means-End Analysis strategy). This sorting is done by a combination of sorting routines: a standard exchange sort for fewer than four rules, an insertion sort for four to fourteen rules, and an iterative version quicksort above fourteen.[11] (Consult ref. 5 or ref. 6 for a detailed description of the various strategies and the reasons for choosing one over the other.) If more than one rule wins under the strategy chosen, a random choice is made to decide the one winner.

Top Level User Interface. The final chapter contains the interface to the user for controlling the OPS5 system, defining the class objects, and specifying the rules. The spirit of Forth is retained as much as possible in that REAL-OPS is a fully interactive,

incrementally compiling version of OPS5. Rules and class objects may be added at any time, the system may be run one or many cycles, and rules may be removed at will. Working memory elements may be made or removed from the top level, and the state of the system may be examined via a set of commands for displaying working memory, matches to condition elements, and the conflict set.

Multitasking and Real Time. Up to this point, the project has been that of rewriting OPS5 in Forth, being careful not to be inconsistent with multitasking needs. Since the goal of this work is to address the needs of the real-time community, the necessary alterations must be made and the interface to external events provided. If an event is expected, it is a simple matter to enter a wait loop, periodically examining an I/O port or a register for the presence of the event. This nice, calm situation obtrains only rarely in the real world of process control and autonomous vehicles, for example. If the event is not expected, how can the "expert" reason about it? Hopefully, the canned expert is not to remain eternally blind, so a way of getting asynchronous (unexpected) events recognized must be found.

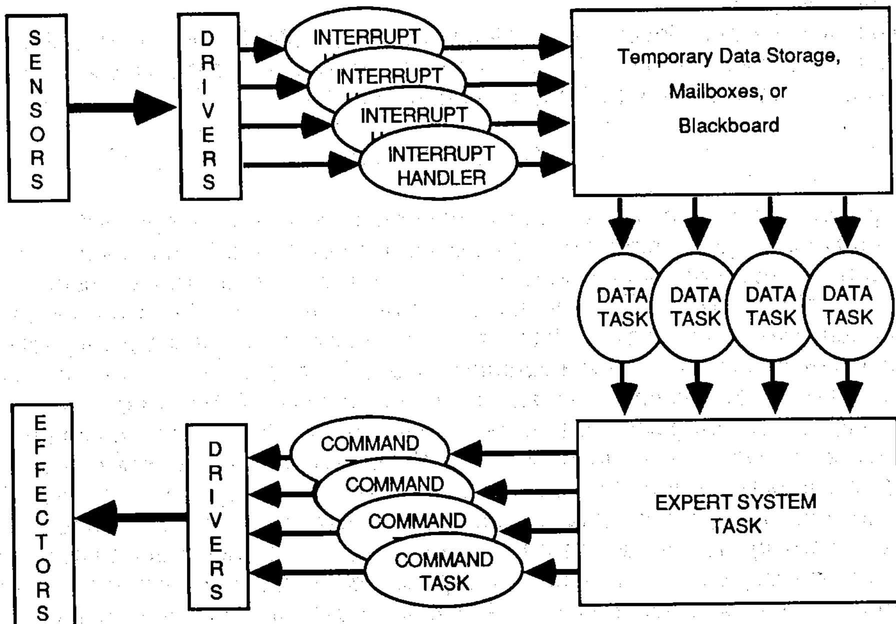

Figure 2. Data flow to and from a real-time expert system.

Figure 2 illustrates a typical block diagram for an interrupt-driven expert system. Sensors continuously provide data about the process or experiment being controlled. The sensor data should provide redundant and multivariate information (one sensor measuring several different properties), perhaps covering different aspects of the same process variable. For example, temperature information could be obtained from thermo-

couples, from radiation measurements in different parts of the spectrum, and from a fiber-optic device. The expert system would be required to "fuse" the various data into a consistent picture about the temperature of the object.

The expert system may be required to take actions (other than **MAKE** and **REMOVE**), changing the course of events in the external world. Sensor readings would presumably change, causing the expert system to respond to the new information. The traditional expert system "knows" where it is going at all times as it is receiving answers to questions it is asking by design. These answers fall into definite, planned categories ("does the animal have feathers?"). Since the system conceived here can neither limit nor know the range or patterns of data that might be presented to it by the real world, external events will alter the course of the patterns of rule firings in ways not forseen by the programmer; events are not synchronous with the recognize-act cycle. Rules should then be robust enough to allow for this eventuality. In a process control problem, the encoded expertise could consist mainly of rules for deciding which control strategies to apply and when the process is at critical points in the multidimensional phase space. A set of goal-oriented rules would provide an overall optimization much as probing is done in control theory.[12]

The next section describes a solution to the problem of communicating external (asynchronous) events to the rules of the expert system. The solution is simple and uses constructs consistent with OPS5 syntax and philosophy.

Using REAL-OPS

The problems of event access described above are recognized by the word "asynchronous," which means the expert system is not "consciously" looking for events, but it must respond to events nevertheless. The method of treating such data is to provide special pathways[13] into working memory. The method can be compared to a once-popular beach toy. The toy had a bucket mounted in a bistable configuration; filling the bucket with sand or water eventually caused the bucket to tip over, spilling its contents into a mechanism of wheels and other buckets. A bucket-defining word was written allowing the programmer to create access pathways into the working memory of REAL-OPS. A bucket looks much like an OPS condition element, but has an attached access-oriented procedure that dumps the contents into working memory when the bucket is filled. Also, a message is sent over the network notifying all interested rules when the working memory element has been made. The bucket vocabulary, shown in Table 1, is typically used from within interrupt service routines or data-handling routines.

An integrated application using REAL-OPS should consist of three parts: the expert system, the real-world interfaces (drivers), and the operator interface for control and display. The combination of buckets and **FORTH** actions on the RHS of rules provides ingress of data into the expert system's awareness as well as carrying out required real-world actions. The ability to call Forth from rules also allows the expert system to access files and databases, plot data in a graphics window, display system warnings through both the graphics and sound mechanisms, and display menus for operator data entry.

Desired explanations of the system's behavior and choices can similarly be enhanced through the use of Forth words graphically displaying the interconnections between rules and condition elements.

Word	Example	Description
BUCKET	BUCKET (METER ^VALUE ^ID ^TYPE)	Creates a new bucket structure called METER, allocating three slots identified as VALUE, ID, and TYPE
PUT	(see below)	Puts the data element into the indicated slot and marks that slot as being filled
INTEGER	INTEGER PUT METER ^ID , or INTEGER 101 PUT METER ^ID	Uses the top item on the parameter stack for a PUT operation, identifying it as an integer. The second method takes the data item from the in-line code for the PUT expression
SYMBOL	SYMBOL Volt PUT METER ^TYPE	Converts the in-line symbol to a token for the PUT operation. A parameter version operates as in the INTEGER case
STRING	STRING " Meter out of service" PUT METER ^TYPE	Converts each symbol in the string into a token, builds a vector in memory, and uses a reference to the vector for the PUT operation. Also has a parameter version as above
VECTOR	VECTOR PUT METER ^VALUE	A reference to a vector is used for the PUT operation. There is also an in-line version as above
DUMP	DUMP METER	Forces a dump of the METER contents into working memory. This is also the demon that dumps a full bucket
FLUSH	FLUSH METER ^VALUE , or FLUSH METER *	Empties the indicated slot (or the entire structure if *). Used to prevent DUMPing if critical data are expected

Table 1. Glossary used by data-handling routines for communicating asynchronous events to the expert system.

Results with REAL-OPS. Real-time benchmarks are difficult to find since each application is different. Falling back on the standard benchmarks of conventional expert systems written in OPS5, two come to mind as having broad appeal: "Towers of Hanoi" and "Monkey and Bananas." "Towers" is a recursive OPS5 program consisting of two rules plus a rule for initializing the towers with rings and one for printing out the results. When printing is inhibited (but the print rule is left in the system), REAL-OPS running on a HP236 (8-MHz MC68000 cpu) runs a seven-tower problem in 25.7 s, while the LISP-based OPS5 takes 2 min on the MicroVAX II, 60 s on a VAX 11/780, and 18.9 s on Texas Instruments' Explorer.

So far, only the sections concerned with matching condition elements to working memory elements and with managing the heap memory pool have been rewritten in 68000 assembler code, leaving much room for speed improvement. The "Monkey"

problem, an enhanced version of the one found in ref. 6, has 30 rules. It runs slightly faster than VAX speeds and is comparable to an optimized C version of OPS5 on the IBM PC. This, of course, provides a hint of where to find additional candidates for code optimization.

A less flashy property of REAL-OPS is its adherence to the spirit of Forth: it is an interactive language. Rules are incrementally compiled as they are entered from the console or from external files. There is no restriction (as there is in OPS5) that all class objects must be defined before any rules may be entered. The only restriction (a very Forth-like one) is that a class must be defined before it can be used in a rule. As mentioned above, rules may be removed (type **EXCISE** instead of **FORGET**) and re-entered. Working memory elements may be entered and removed at will, and the current state of the network and conflict set may be examined (all as in OPS5). In short, developing a program in REAL-OPS has much the same flavor of interactive development as in Forth, much to the user's delight and productivity.

Still to come is a means to "snapshot" a set of rules and working memory in any state of evolution. Also, allowing calls to Forth words on the LHS side of rules will greatly enhance the performance of certain types of rules. Most OPS implementations allow "running the system backward" by keeping up to 32 prior states on a list. This feature makes less sense in a real-time environment where external data might not obligingly repeat itself, but would be of use in debugging, so it may be added in the future.

High-Speed Expert Systems

The main thrust of this paper is to show one way of attaining the goal of very high execution speeds for artificial intelligence, particularly for expert systems, that most recent of AI's applied successes. It has been shown here how Forth can be used to rewrite a successful and popular expert systems language with a resulting improvement in performance and flexibility, as well as extension to handling real-time data. This final section will hint at a means for attaining perhaps a several-orders-of-magnitude improvement in execution speed.

Ideal Forth Engine. From the early days, Forth assumed that the ideal stack-oriented, threaded-code engine was available, and it ran efficiently on this ideal machine. The only trouble was that this engine had to be emulated in the assembly language of the actual processor being used. Upon comparing instruction sets of various processors with those instructions actually used to implement the Forth engine, it was evident that Forth is an efficient utilizer of hardware resources (many machine instructions and registers are never used by the Forth emulator). This observation must have been made by a number of people because there are about a half-dozen architectures providing either an efficient emulation of the Forth engine or actually implementing a two-stack, threaded-code processor.

At Oak Ridge National Laboratory we have been using the NC4000,[14,15] a high-speed Forth engine developed by Novix, Inc. After making several stand-alone work stations[16] based on the Novix Beta Board, we ran a number of benchmarks[17] and ported several programs of interest (including a fast Fourier transform). The memory manager and the list manipulation work mentioned above are operational on the Novix engine, so a test of list management has been performed. The results are sensational

for being done without any optimization, and are presented in a paper by H. G. Arnold[17] but show that the LISP machines are even now outclassed at doing what they do best, namely, list manipulation. The next-generation LISP machines (epitomized by Texas Instruments' MegaChip LISP Machine) will run OPS5 about two to four times slower than a NC4000-based system.

If the current group of Forth engine companies produces a 32-bit version running at 40 MHz (MegaChip's clock frequency), we won't have to tackle the astonishingly difficult and sticky problems of parallel processing for at least a few more years--a serial processor will be (and is!) capable of amazing feats if its natural language is Forth.

Summary

Forth is indeed a language for writing other languages, and it does its job with efficiency of programming, a minimum of code volume, and ease of maintenance. Extensions and modifications to the target language come naturally as needed. The forward-chaining OPS5 provides excellent pattern-matching capabilities for use in an event-driven expert system, and the extensions provided by Forth allow communication to the system from external asynchronous events. This feature is something new for expert systems (even though it has been a bread-and-butter subject for years for many at this conference).

More work needs to be done in REAL-OPS, but it is even now being used.[18] The major new endeavor is to port REAL-OPS to the Novix or Metaforth engines. (There are other Forth engines on the horizon, too.) The time is ripe for Forth-based artificial intelligence applications:[19] the software is here (REAL-OPS, EXPERT-2 to -5 and more,[20] Forth-based Prolog, object-oriented constructs, and so on). The hardware is here, too. The future is exciting!

References

1. See, for example, the review article by Bruce G. Buchanan, "Expert Systems: Working Systems and the Research Literature," *Expert Systems,* Vol. 3, No. 1, pp. 32-51, January 1986.
2. Judith Bachant and John McDermott, "R1 Revisited: Four Years in the Trenches," *AI Magazine*, Vol. 5, No. 3, pp. 21-32, 1984.
3. Peter Hager, "NASA Says AI Systems Just Getting Off the Ground," *Government Computer News* , p. 72, April 11, 1986.
4. Creative Solutions, Inc., *Multi-FORTH Version 2.00 User's Manual*, Rockville, Md., 1984.
5. C. L. Forgy, "OPS5 User's Manual," Technical Report, Carnegie-Mellon University, Department of Computer Science, 1981.
6. Brownston et al., *Programming Expert Systems in OPS5*, Addison-Wesley, Reading, Mass., 1985.
7. Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, Vol. 19, No. 1, 1982.
8. Leo Brodie, *Thinking Forth,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1984.

9. Donald E. Knuth, *The Art of Computer Programming,* 2nd Ed., Vol. 1, *Fundamental Algorithms*, pp. 406-20, Addison-Wesley, Reading, Mass., 1973.
10. W. B. Dress, "A Forth Implementation of the Heap Data Structure for Memory Management,"*The Journal of FORTH Application and Research*, Lawrence P. Forsley, Ed., Vol. 3, No. 3, pp. 39-49, 1986.
11. Donald E. Knuth, *op. cit.,* Vol. 3, *Sorting and Searching*, Chapter 5.
12. O. L. R. Jacobs, "Introduction to adaptive control," *Self-Tuning and Adaptive Control: Theory and Applications,* C. J. Harris and S. A. Billings, Eds., IEE Control Engineering Series 15, Peter Peregrinus Ltd., London and New York, 1981.
13. W. B. Dress, "Communicating Asynchronous External Data to an Expert System," *Proceedings, Eighteenth Southeastern Symposium on System Theory,* IEEE Computer Society, pp. 294-96, April 7-8, 1986.
14. Charles H. Moore and Robert W. Murphy, "Under the Hood of a Superchip: The Novix Forth Engine," Proceedings, 1985 Rochester Forth Conference;*The Journal of FORTH Application and Research*, Lawrence P. Forsley, Ed., Vol. 3, No. 2, pp. 185-88, 1985.
15. Earle Jennings, "The Novix NC4000 Project," *Computer Language*, p. 37, October 1985.
16. R. K. Adams and T. L. Bowers, "Making Novix Beta Boards into Development Workstations," this *proceedings*, 1986.
17. H. G. Arnold, "Symbolic Processing Potential of Forth-Based Microcomputers," this *proceedings*, 1986.
18. James Rash, "A Prototype Expert System in OPS5 for Data Error Detection," this *proceedings*, 1986.
19. See the many papers in this conference proceedings dealing with high-performance hardware architectures and AI-oriented software.
20. Jack Park, "Toward the Development of a Real-Time Expert System," this *proceedings,* 1986.