Constant-time pattern matching for real-time production systems

Dale E. Parson and Glenn D. Blank

Lehigh University, Department of Computer Science and Electrical Engineering
Bethlehem, Pennsylvania 18015

The attached paper appears in *Proceedings of Applications of Artificial Intelligence VII*, Volume 1095, Part 2, pages 971 through 982, edited by Mohan M Trivedi, published by the Society of Photo-Optical Instrumentation Engineers (SPIE), Bellingham, Washington, March, 1989.

Constant-time pattern matching for real-time production systems

Dale E. Parson and Glenn D. Blank

Lehigh University, Department of Computer Science and Electrical Engineering
Bethlehem, Pennsylvania 18015

ABSTRACT

Many intelligent systems must respond to sensory data or critical environmental conditions in fixed, predictable time. Rule-based systems, including those based on the efficient Rete matching algorithm, cannot guarantee this result. Improvement in execution-time efficiency is not all that is needed here; it is important to ensure constant, O(1) time limits for portions of the matching process. Our approach is inspired by two observations about human performance. First, cognitive psychologists distinguish between *automatic* and *controlled* processing. Analogously, we partition the matching process across two networks. The first is the automatic partition; it is characterized by predictable O(1) time and space complexity, lack of persistent memory, and is reactive in nature. The second is the controlled partition; it includes the search-based goal-driven and data-driven processing typical of most production system programming. The former is responsible for recognition and response to critical environmental conditions. The latter is responsible for the more flexible problem-solving behaviors consistent with the notion of intelligence. Support for learning and refining the automatic partition can be placed in the controlled partition. Our second observation is that people are able to attend to more critical stimuli or requirements selectively. Our match algorithm uses *priorities* to focus matching. It compares priority of information during matching, rather than deferring this comparison until conflict resolution. Messages from the automatic partition are able to interrupt the controlled partition, enhancing system responsiveness. Our algorithm has numerous applications for systems that must exhibit time-constrained behavior.

1. INTRODUCTION

The lack of an artificial intelligence programming approach for designing time-constrained, reactive systems has led us to look at the fundamental requirements of these systems. We have found useful design principles by looking at existing, successful systems: people. Humans operate in a wide variety of environments. They can learn to respond to significant, and particularly to dangerous environmental phenomena without engaging in excessive, time-consuming mentation. Practice of consistent activities enhances performance. Finally, humans can usually perform high-level, symbolic processing simultaneously with habitual procedures.

Cognitive psychologists make an important distinction between *automatic* and *controlled* human information processing.[1,2,3,4,5] Automatic processing is any habitual mental or sensorimotor operation. Automatic processes have the following characteristics:[2]

1) The capacity limitations of short-term memory do not hinder automatic processes; these processes do not require attention.

2) A person may initiate some automatic processes, but once initiated all automatic processes run to completion.

3) These processes require considerable training to develop and are most difficult to modify, once learned.

4) Their speed and automaticity will usually keep their constituent elements hidden from conscious perception.

5) They do not directly cause learning in long-term memory, although they can indirectly affect learning through forced allocation of controlled processing.

Controlled processing is closer to the search-based activity commonly studied in artificial intelligence research. Control processes have the following characteristics:[2]

1) They are limited-capacity processes requiring attention.

2) The limitations of short-term memory cause the limitations of controlled processing.

3) Humans adopt control processes quickly, without extensive training, and modify them fairly easily.

4) Control processes direct the flow of information between short-term and long-term memory.

5) Control processes show a rapid development of asymptotic performance.

Qualities 1, 2 and 4 of automatic processes are reminiscent of interrupt handlers in conventional computing systems. Interrupt handlers do not normally require the attention of the processes they interrupt; interrupt handling can be largely transparent to higher level processing that is occurring. Alternatively, because interrupt processing priorities are typically greater than other processing priorities in a system, an interrupt handler has the capability of diverting or preempting the higher level processing when the interrupting event calls for such actions.

People perform many automatic actions in constant-bounded time.[1, Experiment 2] Automaticity fulfills the requirements for reliable reactive processing among humans. It is predictable, efficient, takes priority over controlled processing, and does not consume short-term memory. These characteristics prove to be important features of time and memory restricted, reactive computer processing as well. In the remaining sections we will examine the varieties of constant-time constrained processing for embedded systems. We will illustrate a two-tiered computing architecture for real-time processing, based on the automatic-controlled dichotomy. The requirements and constraints of this architecture translate directly into modifications to the Rete matching algorithm, allowing portions of matching to be performed within predictable time limits. Finally, we will discuss related work and future research problems.

2. A PRODUCTION SYSTEM ARCHITECTURE

2.1 Non-iterative code

The O(1) notation for a segment of code signifies that its time and memory consumption do not grow beyond some predetermined constant limit. The need for predictability in embedded systems requires use of O(1) code for implementation of portions of such systems. We give the name *non-iterative code string* to the form that verifiable O(1) code takes. A non-iterative code string is any piece of code that, at the time it is created, can be rewritten in a form that consists strictly of contiguous, sequentially fetched-and-executed code, combined with forward conditional and unconditional jumps. Iterations bounded by constants can be unwound into non-iterative code. By tracing possible execution paths, a programmer or compiler can calculate worst-case time bounds for non-iterative code. The utility of the notion of a non-iterative code string is that it provides a model of time-bounded computation for design of a programming language and computing architecture. The concept represents a mind-set to assume when examining or building software machinery. The limitations inherent in non-iterative code restrict the range of size and complexity across which input data to the code may vary.

Figure 1 illustrates three ways that non-iterative code fits into an embedded application. *React* code must recognize a significant event within the environment and respond with action within the constrained time. Because the time limitation applies from the instant that sensors register incoming information until the instant that effectors convey reactions, we call this an *instantaneous real-time* requirement. The time restriction applies to the full input port to output port data flow. Success in processing is fully a function of the system's interaction with the environment.

Record code stores incoming sensory data for later analysis. Timing requirements for this processing are simpler than for reactive processing. The instantaneous real-time constraint applies only to capture of incoming information, here in a queue. O(1) computation may store bit patterns as they are read from the input port; some compaction, encoding, or other reorganization might be done. The queue smooths variations in the speed of the incoming information flow. In order to avoid queue overflow, the process draining the queue must consume information at the same average speed as that of incoming information. Therefore, this process must fulfill an *average real-time* requirement. Real-time analyses that focus on the device driver interface typically refer to this type of processing.

Predict processing consists of transmitting predetermined actions to effector outputs. Transferral of output information may be subject to interval timer-based triggering. Like the input portion of record code, the output portion of predict code (after the queue in Figure 1) is bounded by constant time limits.

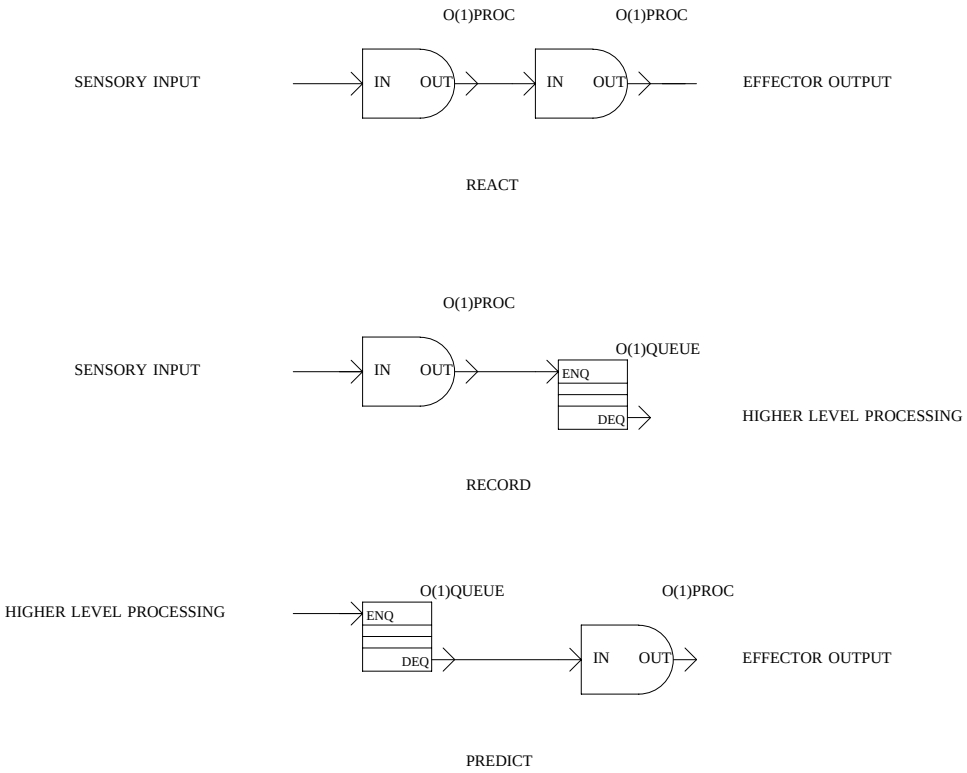

Figure 1 - Three varieties of constant-time-constrained processing

Note that since the time requirement on react code is not averaged across multiple incoming events, there is no need for dynamically varying buffers in a react code sequence. If an incoming datum is being processed so slowly that a second incoming datum demands processing resources before the first releases them, then the time constraints met by the reactive code are insufficient for the frequency of input arrival. In instantaneous real-time processing, the required ratio of processing speed to input arrival speed must be maintained on a per input event basis. Besides simplifying memory management and timing analysis, this arrangement is attractive because it matches the characteristics of automatic processing very closely. It satisfies characteristic 1: the reactive code does not consume limited short-term memory for buffering. It satisfies characteristic 4: the lack of short-term memory results in a lack of any tracing ability for reactive processing; there is nowhere to store information about internal reactive processing, so its history is reflected only in the actions it performs within the environment. It satisfies characteristic 5: the lack of short-term memory results in the absence of any staging area into long-term memory for learning. This arrangement implies characteristic 3: since declarative memory of the internals of reactive processing is unavailable for the refinement of this processing, the environment itself must act as storage. Learning automatic reactions requires practice because the reactive code stores responses to approximations of the environment; practice brings these responses into contact with their target phenomena so that learning may refine the responses a little at a time. A reactive system does not rely exclusively on planning, but practices automatic processes because the complexity of the environment would exhaust planning storage capacity. Finally, reactive processing avoids buffering because the input ports in an embedded system represent immediate information in the environment. When an input sensor registers a new sensum, the system considers any prior sensum outdated (at least for purposes of immediate response) and discards it.

2.2 A two-tiered architecture

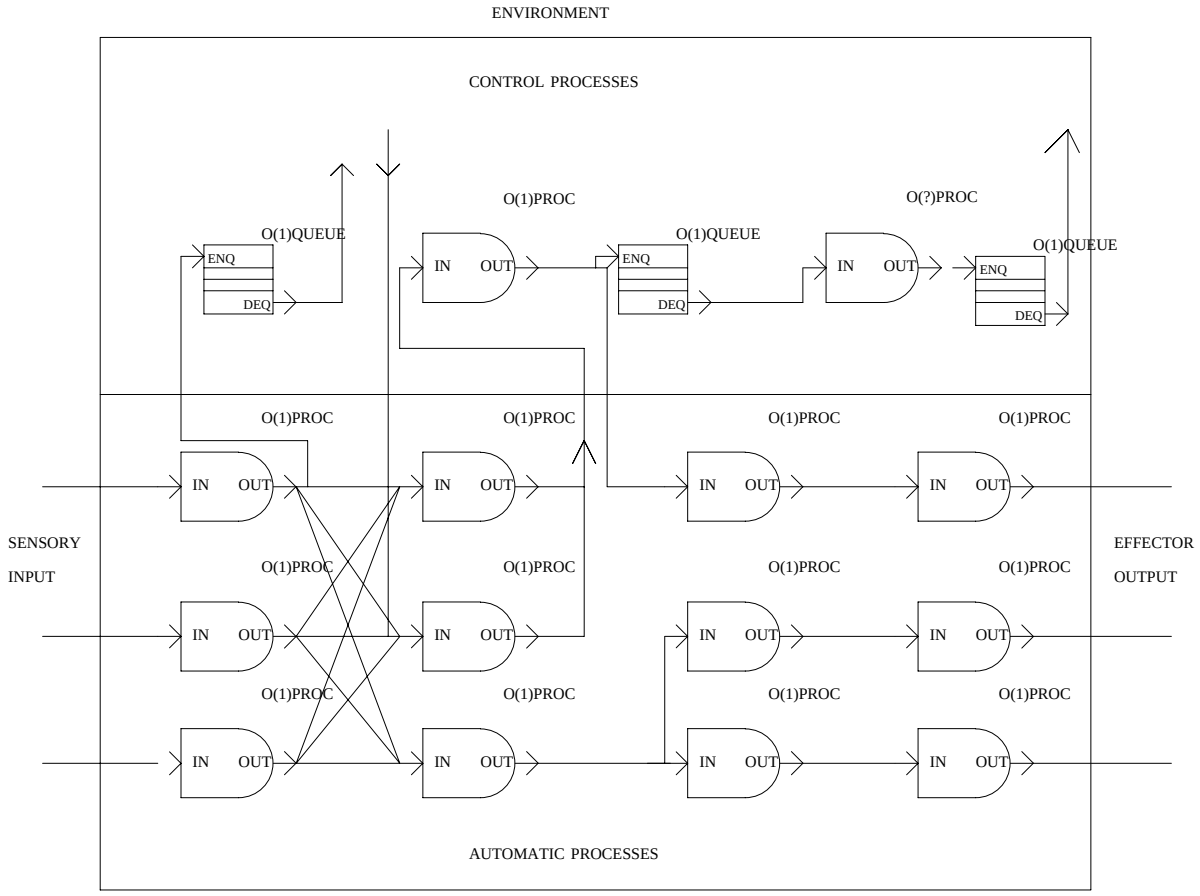

Figure 2 - The controlled-automatic architecture

Figure 2 illustrates the essentials of our two-tiered architecture. The automatic partition is in the bottom half of the figure. It is composed of non-iterative processes that are connected into non-iterative combinations within the partition. The data flow through the automatic partition can be represented as a directed acyclic graph. The only feedback loop in the automatic partition is an inherent one, mediated by the external world. It goes from effectors through the environment to the sensors. (Internal feedback loops are of course possible in the controlled partition.) The leftmost level of automatic processes serve to condition data they accept from the sensors. Combinations of conditioned sense data are fed to processes that might represent feature detectors. Time constrained decoding of input is important for the recognition of significant environmental conditions. In a conventional computer system, incoming interrupt lines often enjoy a one-to-one correspondence with important conditions which must be attended. Essential feature detection and recognition is hard-wired. In an intelligent system, recognizing complex phenomena requires programming or learning. The input decoding network (software) must take on part of the job of interrupt generation.

After decoding and code conversion, automatic processes may feed their output forward to mechanisms that drive effectors. Two partially shared paths from sensors to effectors are shown at the very bottom of Figure 2. All reactive processing - processes for which the port to port processing time is constant bounded - conform to this format.

In addition, automatic processes may feed information into the controlled partition and receive information from there as well. Typically we will design the controlled partition so that it will be responsive. For example, it might complete the particular process upon which it is working, and then respond to important incoming information from

the automatic partition. The environment may establish a context for goal-directed controlled processing, and when this context ceases to exist, the controlled processing should recognize this change and perhaps suspend the goal-directed activities. We do not, however, guarantee that this response will meet time constraints.

Data flow from the controlled to automatic partition includes enabling and disabling of automatic sequences. According to Schneider and Fisk,[3] controlled processing affects skilled performance in three ways. First, controlled processing maintains strategy information in short-term store to enable sets of automatic productions. Second, controlled processing maintains time-varying information in short-term store. Third, controlled processing in skilled behavior manages problem solving and strategy planning. Reliance on controlled processing for maintenance of short-term storage underscores the lack of persistent declarative memory in the automatic partition.

2.3 A programming notation

Forward-chaining production systems have a stimulus-response organization. They represent a form of knowledge representation that is closer to the idea of non-iterative code strings than other artificial intelligence programming architectures. The patterns of interconnection of nodes in semantic nets often result in exponential search time for approaches such as spreading activation; also, semantic nets may contain cycles. These characteristics are shared with frame systems; in addition, the latter may contain procedural attachments of arbitrary computational complexity. The computational problems associated with first order logic as a representation language are well known. In contrast, individual productions in a forward-chaining production system do not iterate. Iteration is achieved through composition of productions, when data flow dependencies among productions form a cycle. A compiler can readily detect iterative composition of productions.

Unfortunately, the algorithms implementing the run-time environments for production systems do not guarantee constant-time bounded responsiveness. The Rete pattern matching algorithm[6] is a popular and efficient approach used to match working memory changes to changes in the conflict set of instantiated productions. However, Rete matching can consume time that cannot be predicted when the program is compiled. Production systems typically spend the majority of their execution time performing pattern matching.

We have adopted the production system notation and have augmented it with some novel, time-constrained semantics. In the remainder of this paper we will discuss the implementation of a two-tiered production system architecture, and will examine how matching in the automatic partition departs from the methods of Rete.

2.4 An example Rete network

Numerous discussions of Rete and its application to production system implementation exist.[6,7,8,9] Introductions to OPS5 are also available.[10,11] Rather than repeat the discussions of Rete within OPS5 here, we will start by giving an example of Rete's implementation of a simple set of OPS5 productions. We will follow this by examining corresponding productions and a matching strategy for a PRIOritized Production System (PRIOPS). The architecture of PRIOPS is based on the automatic-controlled processing dichotomy.

Suppose that an OPS5 program contains the following two productions. The first, **overtemp**, detects when a temperature sensor reading has exceeded a limit previously stored in working memory. This production reacts to this condition by making an *alarm* memory element to record the event, and an *action* memory element to initiate a reaction. The action memory element triggers another production (not shown), which actually turns on the effectors. In this case, the effector action is more important than recording the alarm. To get this preference, production **overtemp** exploits OPS5's *recency* criterion, generating the more urgent action memory element last.

The second production, **anytrace**, records all new information from sensors of a specific type. The sensor type is selected by memory elements matching anytrace's first condition element. It might supply environmental trace information to a learning algorithm.

Figure 3 illustrates the Rete net produced by these two productions. On the left side are two columns of *test* nodes leading down to the *join* node, <HITEMP> > <HILIM>. They represent the matching required for the two condition elements of production **overtemp.** The leftmost column tests for a working memory element of class "limit" whose "type" value is "temperature." What Rete does is store pointers to "limit" elements that satisfy these tests in local

```
(p overtemp
        (limit ^type temperature ^hilimit <hilim>)
        (sensor ^type temperature ^id <tempid> ^direction <way>
                ^reading {<hitemp> > <hilim>})
        -->
        (make alarm ^type hi-temp ^limit <hilim> ^reading <hitemp>)
        (make action ^type move ^direction (opposite <way>)
                ^speed fast)
)

(p anytrace
        (trace ^type <tracetype>)
        (sensor ^type <tracetype> ^id <sensid> ^reading <value>)
        -->
        (make tracer ^type <tracetype> ^id <sensid> ^reading <value>)
)
```

memory associated with this pre-join chain. The second column are the tests for the second condition element of **overtemp.** The two columns converge in a *join* node that tests the inter-condition element dependencies for this production: comparing <HILIM> from "limit" with <HITEMP> from "sensor". Whenever this join test succeeds, Rete modifies the conflict set.

The rest of the Rete net matches production **anytrace.** Note that the two productions share a reference to a "sensor" element. Rete exploits this commonality. The "sensor" test for **anytrace** is simpler than for **overtemp,** not needing a match on the "type" field. Consequently the flow of "sensor" tokens for **anytrace** diverges from **overtemp** tokens before the latter enter the type test. Token flow for other "sensor" condition elements that test "sensor" differently would also diverge at this point. Standard Rete traverses each of these paths, at least until the token fails a test, possibly as far as the conflict set.

As Haley[12] has pointed out, we can compute weak worst-case execution times for pre-join portions of a Rete net at compile-time. The most pessimistic estimate involves summing the time to execute all pre-join tests for a specific working memory element class each time such a working memory element is added or deleted. Better estimates can be obtained if we can determine that certain paths are mutually exclusive in the elements they will match. Such exclusion can certainly be determined when all successors to a node within a pre-join chain test for distinct, constant values of a single field, or for non-overlapping numeric ranges in a single field. Unfortunately, the lack of field typing and the use of LISP-like symbolic atoms for field values preclude the possibility of determining mutual exclusion for negated constant tests. We will address this issue in the section on PRIOPS. We will also will show a way to use production priorities to defer portions of pre-join matching.

Haley[12] has also shown that the time to compute joins dominates the matching time of Rete. Worst-case join times cannot be determined at compile-time because the sizes of Rete memory nodes are not restricted at compile-time. In the example of Figure 3, the first memory node might hold many "limit" tokens; Rete places no restriction on the number of working memory elements of a class. A "sensor" token entering the right side of the first join node must be tested against all "limit" tokens stored to the left. Join time is therefore a function of both the history of pre-join adds and deletes, and the tests performed within the node. This history is unknowable at compile-time. Haley suggests several generic approaches for restricting the size of memory nodes contributing to joins. We apply a specific restriction to memory sizes in the automatic partition, along with priority-based deferment of join matching, in the next section.

2.5 The PRIOPS design

The notation of the PRIOPS language is very similar to that of OPS5. PRIOPS' method of using production priorities is the key feature that distinguishes it from other production systems. A programmer can explicitly assign a priority from -128 to 127 to each production. The default priority is 0. The main function of the priority value is

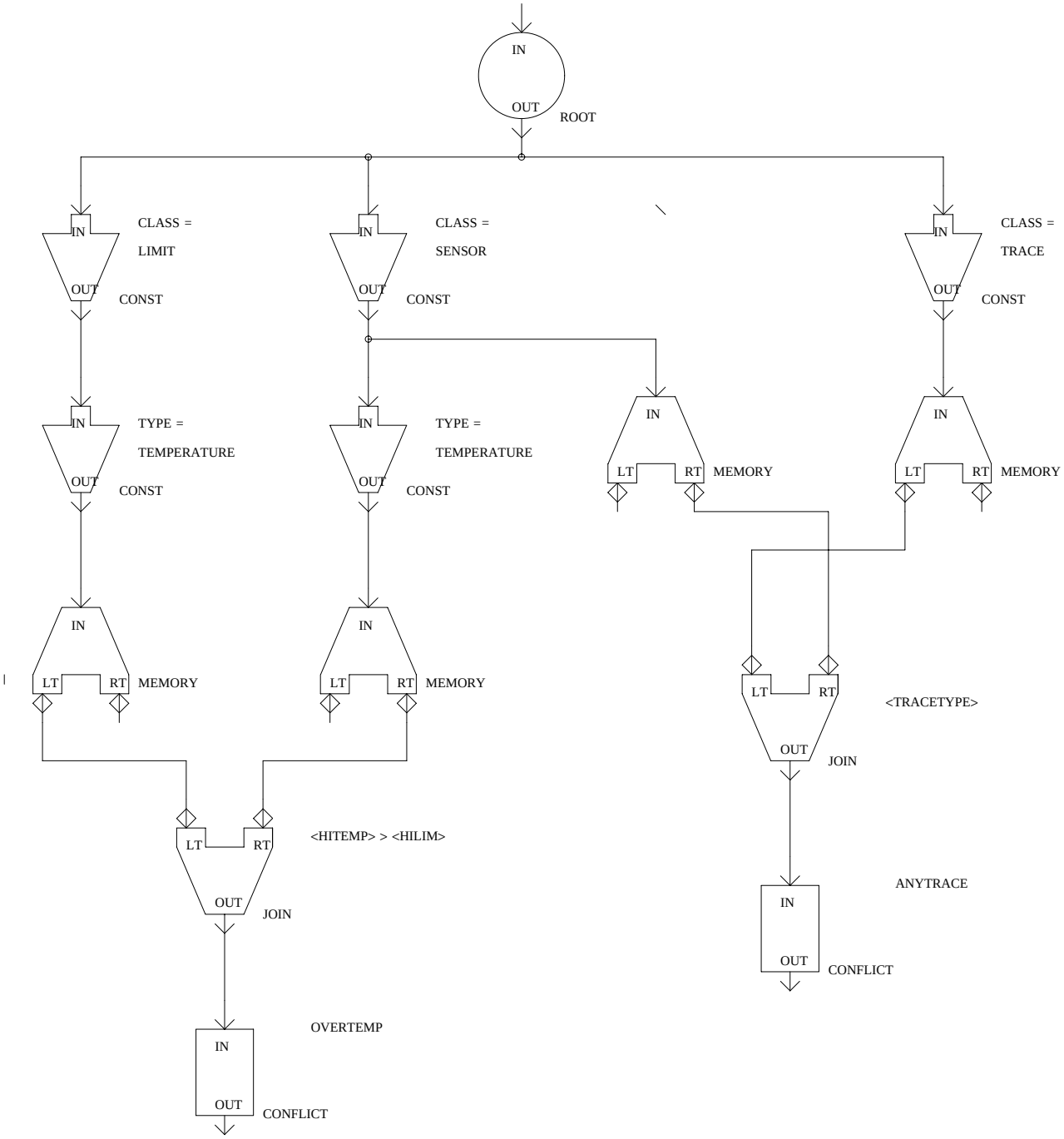

Figure 3 - An example Rete network

assigning the matching network derived from the production's left hand side to the automatic or controlled matching partition. Productions with priorities > 0 are automatic productions; their matching network is part of the automatic partition. The remaining productions contribute tests to the controlled matching partition.

The controlled matching partition is similar to the standard Rete matching network. The automatic partition, on the other hand, differs from standard Rete in several important ways. First, production priorities allow postponement of matching actions for lower priority productions in deference to higher priority productions. Subsequent actions by high priority, executed productions may cancel these deferred lower priority matching processes. Second, the size of memory nodes within the automatic network does not exceed one. Restricting the size of automatic memory nodes yields maximum join times that can be computed as a function of the join tests. This time can be calculated when the production is compiled. Third, the compiler can perform data flow analysis for the automatic partition, detecting any iterative composition (and corresponding execution time indeterminacy) of automatic productions. An example will help in elaborating these points.

2.6 Example PRIOPS productions and matching network

Below are two PRIOPS productions which correspond to the OPS5 productions discussed above. The resulting network is shown in Figure 4.

```
(p overtemp-on-t1 1
        (limit ^type temperature ^target (contains t1) ^hilimit <hilim>)
        (sensor ^type temperature ^id t1 ^direction <way>
                ^reading {<hitemp> > <hilim>})
        -->
        (make alarm ^type hi-temp ^limit <hilim> ^ reading <hitemp>)
        (make action ^type move ^target wheels
                ^direction (opposite <way>) ^speed fast)
)

(p anytrace -1
        (trace ^type <tracetype>)
        (sensor ^type <tracetype> ^id <sensid> ^reading <value>)
        -->
        (make tracer ^type <tracetype> ^id <sensid> ^reading <value>)
)
```

PRIOPS requires strict typing of memory element fields, a restriction unknown to OPS5. Possible types include symbolic atoms, integers, floats, and symbol sets. The latter are similar to Pascal sets,[13] and are implemented as bit-vectors. Operations on bit-vectors complete in constant-bounded time, determined by the vector's maximal size. Symbols sets are used to aid in the determination of mutual exclusiveness of tests; when the universe of a set is enumerated at compile-time, both negative and positive tests identify potential successful values. Strong typing is not an O(1) consideration, since type tests consume constant-bounded times; compile-time typing is included for the sake of run-time efficiency. Working memory classes are like Pascal record types in that only fields declared with a class may be accessed within a memory element. The compiler can determine memory element sizes, thereby accommodating constant-time bounded allocation algorithms.

Several notational differences distinguish these PRIOPS productions from their OPS5 counterparts. Each production carries an associated priority, 1 for **over-temp-on-t1,** -1 for **anytrace.** The *contains t1* test in the first production is a test for membership of "t1" in a symbol set field. Perhaps the most conspicuous difference is the fact that the first production has become specific to a single sensor. The reason for this specialization is that for all automatic productions, the output from all pre-join test chains and join test nodes are deposited in memory nodes that can hold only one item each. Therefore it is necessary to create distinct, specialized productions for distinct sensor readings or other distinctly identifiable data that must be saved. The single-item memory nodes implied by a distinct production's condition elements will be created for the corresponding distinctive data. A macro facility can generate a collection of similar productions specialized for uniquely identified data. Note that similar pre-join chains need only

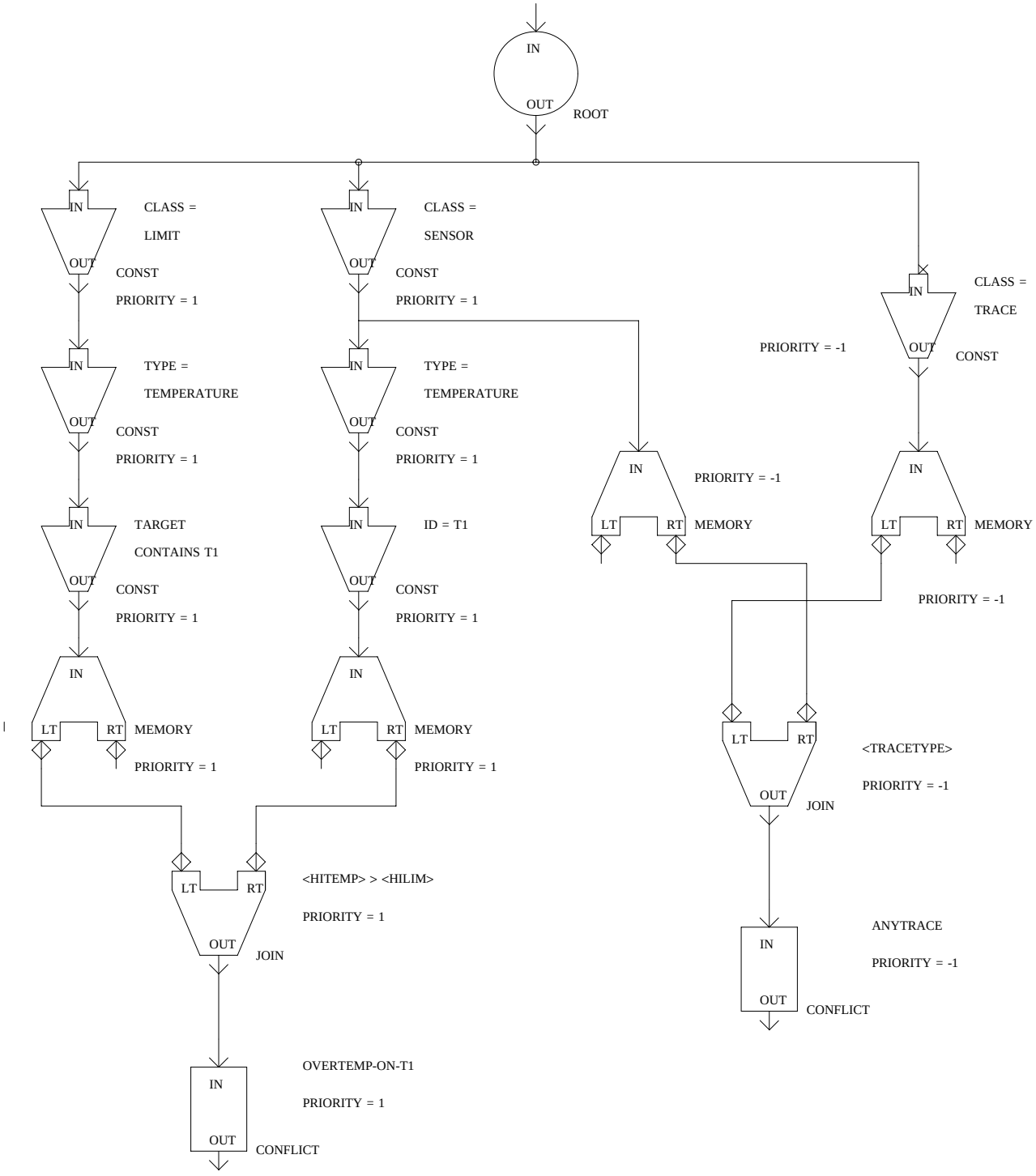

Figure 4 - An example PRIOPS network

diverge at the point at which this distinguishing identification field is tested. This test should be placed after all shared tests, allowing the majority of pre-join tests to be executed once per working memory change. The executable code for join nodes that differ only in their input and output memories can be shared.

The restriction of unit memory node size is the most significant characteristic of the automatic partition. The rationale is that the majority of automatic tokens represent sensory data or direct derivatives of sensory data. Lack of persistent memory results in contents for these one-place buffers that reflect the current state of the environment. This restriction on size accords well with observations on human automatic processing as already discussed. With history removed from the sense-decoding, reactive partition, join times are a function of join tests. In a seminal paper on production systems, Newell questions the very name "short-term memory," since this temporary activity is so very limited in capacity.[14] Working memory in early production systems was intended to hold transient data during matching, not long-term control and application domain knowledge. While modern production systems have often strayed far from this original, restricted notion of a limited short-term memory, the concept is fundamental within the automatic partition of PRIOPS. Because automatic memory nodes are of such limited capacity, it is necessary for the compiler to generate distinct memory nodes for each sensor to hold sensor-based data. Thus while memory nodes diminish in size within the automatic partition, mutually exclusive test paths proliferate. Since these paths are mutually exclusive, execution time reactivity is enhanced. Sensor-based token propagation corresponds to OPS5 *modify* operations, since old information is deleted and new is added.

Comparing Figures 3 and 4, we find the "limit" pre-join nodes filtering tokens for "t1" containment. Other condition elements for other sensors would diverge at this point. The set notation obviates the need to generate a distinct "limit" memory element for each sensor. A single "limit" element assertion can enable a group of overtemp productions.

Note that the "limit" test nodes have a priority of 1. A node's priority is the maximum priority of the productions that share that node in their left hand side tests. It is possible for automatic and controlled productions (> 0 and <= 0 priorities) to share pre-join nodes; memories and post-memory nodes are strictly partitioned. Unlike production systems which utilize priorities only at conflict resolution time, PRIOPS employs them during matching. Matching sets the current system priority to the priority of the firing rule as its right-hand-side actions begin. Right-hand-side generated working memory changes propagate through the network in priority order for changed memory elements whose priorities are > 0. As matching is performed, paths with priorities lower than the current priority are placed at the tail of a double-ended queue for that priority. When matching encounters multiple paths at the current priority, all but one are placed at the front of the queue for that priority. Finally, when a path with a priority greater than the current priority can be satisfied, the old path is placed in the front of its queue, and the higher priority becomes the current priority. Matching traverses all paths at the highest current level, placing all triggered productions with the highest current priority into the conflict set. *Automatic partition conflict resolution* considers *priority* first, *age* of contributing memory elements second, and *specificity* last. When instantiation priorities are equal, conflict resolution considers age in an attempt to fire sensor-triggered rules before reaction time constraints are exceeded. Unlike OPS5's *recency* criterion, older information receives preference since it may soon become outdated for reactive use. Conflict resolution for production instantiations at the current priority occurs after the current priority queue is emptied, without considering lower priority pending computations. Sensor interrupt handlers only advance their readings when either a) they have emitted no prior readings, in which case their readings are attached to the tail of their priority queue; or b) their prior readings are part of the current matching. In the latter case current readings are not advanced until the current priority falls below their priority. Some minor buffering within sensor interrupt handlers may be appropriate, but repeated loss of sensory data signifies excessive time consumption by equal and higher priority processes.

When no automatic productions are instantiated, matching of controlled productions ensues. We assume the availability of a single processor here; multiprocessing could allow for simultaneous operation of the two partitions. Inference does not use controlled production priorities until conflict resolution. Conflict resolution for controlled productions uses variants of OPS5's MEA and LEX strategies, with priority used as the primary criterion. Recency, and therefore history, becomes significant in controlled conflict resolution. The controlled Rete net does not restrict memory sizes and join times. Each step in controlled production matching is bounded by constant time, however, and controlled matching can be preempted at the end of the current step. Preemption can occur after each step of a join

process, with the current join process placed at the head of the controlled partition's matching queue.

Matching avoids critical section problems by performing node tests in small, uninterruptible, constant-time bounded increments for both partitions. Constant-time bounded memory allocation is used, and explicit recovery takes the place of garbage collection. Still, the possibility exists that an action at a lower priority - say the addition of a working memory element - could trigger a higher priority production which would cancel that original action - say remove the element - while parts of the original action are still pending in queues. To avoid the specific problem of removes getting ahead of the adds that triggered them, we use Gupta's *extra-deletes-list* [9, p. 67] to cancel outdated adds in the controlled partition. A remove action that does not find the token it is supposed to delete, is deposited within this list at the memory node. When the corresponding add eventually arrives at the memory node, it finds the waiting delete and neither action advances further in the net. This technique is not needed in the automatic partition, where any new add or delete determines the entire contents of the unit size memory. Only one matching action need be queued along an automatic path; any new data advancing down the same path replaces any prior data.

Getting back to Figure 4, initialization can propagate "limit" information prior to performance time unless limits are acquired during execution. Matching considers "sensor" information first for priority 1, and if **overtemp-on-t1** fires, then the -1 priority path for "sensor" remains in its queue. Furthermore the actions of **overtemp-on-t1,** the "make alarm" and "make action", are performed according to priority. Therefore the higher priority action - initiating the movement action - is performed at the earliest possible time (we are assuming that one or more productions triggering on "action" have higher priorities than those triggering on "alarm"). The result is that high priority reactions do not wait for lower priority matching. In addition, high priority actions may change the environment sufficiently so that low priority operations can be cancelled before their matching is completed. Constant-time bounds can be computed by adding worst-case matching delays at a given priority, and the sum of such delays for all higher priority paths times the frequency of triggering (sensor triggering frequency) for these paths.

2.7 Related work and future research

Related work on partitioning production memory has been performed in the PAMELA system,[15] with its use of *DEMON productions* that fire immediately upon satisfaction. Our approach differs in its restriction on automatic memory sizes and join times, its ordering of right-hand-side memory change actions by priority, its use of priorities for scheduling at matching time, and the conflict resolution strategy for the automatic partition. The PAMELA system does not impose constant limits on memory node sizes and does not use priorities to defer portions of matching for a single memory change. Each working memory addition, deletion, or modification in PAMELA is an atomic action. Conflict resolution on the high priority DEMON conflict set occurs at the completion of each working memory change. PRIOPS working memory changes are not atomic; low-priority portions of the resulting matching are deferred. Also PAMELA requires explicit calls to synchronization tests for rules that may be interrupted by DEMONS that remove or modify memory elements bound within the rules. In PRIOPS, any interrupted controlled rule instantiation removed from the conflict set due to automatic actions, is terminated and its modifications to working memory are retracted. Retraction is performed by countering the rule's adds with deletes, and the rule's deletes with redundant adds of the deleted working memory elements. Scheduling places retraction actions at the head of their queues, ahead of the operations that they will cancel. Redundant adds propagate normally, unless they find earlier adds not yet cancelled because a delete is queued. Any add that finds that it is redundant at a memory node, deposits itself in an *extra-adds-list*, the counterpart of the extra-deletes-list mentioned earlier. Deletes first inspect a memory node's extra-adds-list, and if they find that they have in effect been cancelled, they consume the extra add and terminate. Because both extra lists will typically be empty, these tests are fast. They are not employed within the automatic partition, where rules respond primarily to sensory data; once triggered, these execute to completion in accordance with the principles enumerated on the page one. Consequently PRIOPS automatic rule execution is atomic but constant-time bounded. Controlled rule execution appears atomic to the programmer, but is in fact preemptable and retractable.

Additional synchronization of the controlled partition with the automatic partition is possible. For example, environmental conditions might spawn controlled, goal directed activities. The high temperature condition in our earlier example might activate two goals, one to discover action sequences to avoid this undesirable condition in the

future (e.g., "stay away from the fire"), and another to search for ways out of the present situation (in case automatic reaction fails, perhaps due to a failed effector). The latter goal has higher immediate priority. However, if automatic reactions do alleviate the high temperature problem, then only the former goal need be pursued; the latter has become outdated. Mechanisms for detecting when environmentally inspired controlled activities (such as the latter goal) should be discarded are currently designed ad hoc. Creation of more sophisticated automatic-controlled synchronization as part of the production system notation and architecture is an area for future research.

3. CONCLUSION

PRIOPS is still in the iterative design, experiment, and modify portion of its life. Initial applications include paper and toy problems, such as an interactive video game that learns responses. The architecture appears very promising, and work in the area of learning and practice is planned. We believe that many practical control problems are amenable to solution using this dichotomous approach.

4. REFERENCES

1. Walter Schneider and Richard M. Shiffrin, "Controlled and Automatic Human Information Processing: I. Detection, Search, and Attention," *Psychological Review,* Vol. 84, No. 1 (January, 1977), p. 1-66.

2. Richard M. Shiffrin and Walter Schneider, "Controlled and Automatic Human Information Processing: II. Perceptual Learning, Automatic Attending, and a General Theory," *Psychological Review,* Vol. 84, No. 2 (March, 1977), p. 127-190.

3. Walter Schneider and Arthur D. Fisk, "Attention Theory and Mechanisms for Skilled Performance," *Memory and Control of Action,* ed. Richard A. Magill. Amsterdam: North-Holland Publishing Co., 1983, p. 119-143.

4. Richard M. Shiffrin and Susan T. Dumais, "The Development of Automatism," *Cognitive Skills and Their Acquisition,* ed. John R. Anderson. Hillside, NJ: Lawrence Erlbaum Associates, 1981, p. 111-140.

5. Arthur D. Fisk and Walter Schneider, "Memory as a Function of Attention, Level of Processing, and Automatization," *Journal of Experimental Psychology: Learning, Memory, and Cognition,* Vol. 10, No. 2 (April, 1984), p. 181-197.

6. Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem," *Artificial Intelligence* 19 (1982), p. 17-37.

7. Charles L. Forgy, *On the Efficient Implementation of Production Systems.* Department of Computer Science, Carnegie-Mellon University, January, 1979.

8. Daniel J. Scales, "Efficient Matching Algorithms for the SOAR/OPS5 Production System." Report No. STAN-CS-86-1124, Stanford University, June, 1986.

9. Anoop Gupta, *Parallelism in Production Systems.* Los Altos, Ca: Morgan Kaufmann, 1987.

10. Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin, *Programming Expert Systems in OPS5: An Introduction to Rule Based Programming.* Reading, MA: Addison-Wesley, 1985.

11. Charles L. Forgy, *OPS5 User's Manual.* Memo CMU-CS-81-135, Carnegie-Mellon University, July, 1981.

12. Paul V. Haley, "Real-Time for Rete." *Proceedings of ROBEXS '87: The Third Annual Workshop on Robotics and Expert Systems,* Research Triangle Park, NC: Instrument Society of America, 1987.

13. Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report,* Second Edition. New York: Springer-Verlag, 1974.

14. Allen Newell, "Production Systems: Models of Control Structures," *Visual Information Processing,* ed. William G. Chase. New York: Academic Press, 1973, p. 523-524.

15. Franz Barachini and Norbert Theuretzbacher, "The Challenge of Real-time Process Control for Production Systems." *AAAI-88, Seventh National Conference on Artificial Intelligence,* Volume 2. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1988, p. 705-709.