--

--

Automatic versus controlled processing: an architecture for real-time production systems

Dale E. Parson and Glenn D. Blank

Lehigh University, Department of Computer Science and Electrical Engineering

Bethlehem, Pennsylvania 18015

This paper appeared in the *International Journal of Expert Systems: Research and Applications,* Volume 2, Number 3/4 (1989), ISSN 0894-9077, p. 397-422.

Automatic versus controlled processing: an architecture for real-time production systems

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, 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.

Key words: real-time, production system, Rete algorithm, embedded system, reactive processing.

1. INTRODUCTION

Human intelligence acts within the context of sensorimotor interaction with a physical environment. While a person may process considerable information that is independent of the immediate situation - including planning for the next day, planning for upcoming years, worrying about problems, and fantasizing about prospective achievements - the world does not pause and allow this processing to proceed uninterrupted. The oncoming car preempts the driver's daydream. The crying baby interrupts the parent's reading. The human must respond to environmental conditions, often in predictably constrained time. When ongoing thoughts and emotions compete with urgent physical demands for attention and response, the urgent demands win. Fortunately for our concentration, many time-constrained interactions with the physical world are not preemptive; while response may be important or even critical, the nature of the response is straightforward and stereotyped. Experience with the environment - practice - builds a repertoire of habits. Though an individual habit is inflexible and restricted in its range of applicability, a coordinated collection of habits insulates the higher order cognitive system from many of the demands on processing made by the interacting world. Habitual activity *is* a form of processing, one which differs both qualitatively and quantitatively from more complex cognitive activities such as problem solving, planning and learning. It is simple, efficient, and occurs in multitudinous instances.

Existing artificial intelligence processing architectures are predominately complex organizations designed to address the needs of higher order cognitive activities. In this paper we present a two-tiered computing architecture, one based on an existing model of human information processing. We concentrate on the layer corresponding to habitual activity in people. First we examine the underlying psychological theory. Then we present a mapping of this theory into the domain of reactive computing. Finally we discuss a production system called *PRIOPS* (PRIOritized Production System) that embodies this two-tiered architecture.[11] PRIOPS employs a priority based variant of the well known *Rete* matching algorithm.[4,5] This final section includes discussion of related work and future research

problems.

2. CONTROLLED AND AUTOMATIC HUMAN INFORMATION PROCESSING

Schneider and Shiffrin first proposed their theory of *controlled* and *automatic* human information processing in 1977.[14,16] Controlled processing is the complex cognitive activity often modelled in artificial intelligence programs. It is flexible, is capable of problem solving and learning, but is serial in nature and inefficient when dealing with reactive problems. Automatic processing is the formal name for habitual perceptual and cognitive activity. Schneider and Shiffrin originally applied this two-level architectural theory to human performance on practiced recognition and recall tasks; later research dealt with practice and skilled performance of sensorimotor tasks.[13] An overview and general definition of automatism is found in Shiffrin and Dumais.[15] Here we present the important characteristics that distinguish controlled and automatic processing.

2.1 Attention and memory

Controlled processing requires *attention.* In this theory, attention is a serially reusable, limiting resource. Attention acts as a bottleneck; as a result, controlled processing is serial. Automatic processing, on the other hand, does not require this serial resource. Automatic processes are free to act in parallel, at least in cases where they do not conflict or compete.

Controlled processing also requires use of *limited short term memory;* automatic processing does not. Attention directs controlled processing, while short term memory records its state. Does this imply that automatic processing is undirected and unrecorded? In fact, automatic processing is directed by the contents of the sensory field in contact with accumulated experience; accumulated experience is long term memory achieved through repetitive practice, so automatic processing is a form of reactive long term memory. Automatic processing is not directed through the focusing of higher, problem solving attention, but through the reliving of past experience in the present environment. Automatic processing does not require explicit state recordings for its examination.

An example is in order. When you first learned to tie your shoes, you used controlled processing. Attention directed the activities of the novel task, relying on short term memory to provide information about how to cross the shoestrings. Short term memory also stored the intermediate states of the tying process, so you knew what to do next at each stage. After many shoe tying episodes, however, you committed the complete sequence to long term memory. This was possible because the complete sequence was invariant and therefore predictable. You did not need to use attention to search for each step, because each intermediate step had been memorized. You did not use short term memory to record each intermediate state as it occurred, because you knew all intermediate states well in advance. The parallel activity of automatism is limited by the sensorimotor system; since the hands cannot tie shoes and turn pages at the same time, there is a sensorimotor bottleneck. You can, however, look at, think about and remember something else while tying your shoes. You may not remember stopping to tie your shoes at all. Automatic processing handles the shoe tying, while controlled processing proceeds independently.

2.2 Interruptions and priority

Circumstances arise where automatic processing forcibly redirects controlled processing's attention. Indeed Shiffrin and Dumais define automatic processing as including any activity that does not consume attentional capacity, or that always consumes attentional capacity whenever a given set of external initiating stimuli are present, regardless of the person's attempt to ignore or bypass the distraction.[15] The second, alternative characterization leads us to a notion of automatic processing as prioritized interrupt handling. It is prioritized because it is given greater priority than controlled processing in situations that *must* be attended; it is capable when necessary of preempting and diverting the flow of controlled processing. From the perspective of controlled processing, automatic processing is seen as atomic. Once initiated all automatic processes run to completion, but their speed and automaticity usually keep their constituent elements hidden from conscious (controlled) perception.[16] The idea of automatic processing as complex interrupt handling capable of operation independent from controlled

processing, and capable of redirection of controlled processing in urgent situations, is fundamental to the use of priorities to focus matching and reactive activities in PRIOPS.

2.3 Practice and learning

Habits are inflexible activities that require consistent practice to develop, operate efficiently within the applicable situations, and are difficult to ignore or abandon. In comparison, analytic thought is flexible, slow, and at times creative. The degree to which a person can learn automatic responses is directly related to the degree to which he/she can practice these responses *in situ.* [12] The consistency is important for our research because it is a prerequisite to *predictability.* If a processing system must react to an important environmental situation within predictable time, then the environmental situation must itself be predictable; only then can a predictable response (and response time) be determined. Practice collects information about predictability of environmental conditions. Variants within these situations will *not* be automated; they will require the search focusing and short term state saving capabilities of controlled processing. Consequently any complex situation will require a mix of controlled and automatic processing. Only the latter, however, will be predictably responsive; the responsiveness of controlled processing, because it deals with unpredictable and novel stimuli, will be largely unpredictable. Some combinations of automatic and controlled processing may be *statistically* predictable, especially when automatic processing dominates.

With the emphasis on planning that exists in artificial intelligence, a question arises: why should people acquire automatic processing exclusively through practice? Practice collects information about the consistency of elements of environmental situations. If reliable information about a situation is available without practice, why should a planning process be unable to generate a reliable plan for time-constrained reactive processing? We believe that the answer is, for any realistically complex environmental situation, that short term storage capacity is insufficient to hold all of the information necessary for *a priori* planning of a complete sequence of reactions. Practice makes modest, immediate demands on short term store. The environment itself acts as a reliable long term store; practice searches

this store a region at a time, and the importance of a region is determined as it is examined. For demanding situations, practice can proceed in a watered-down version of the eventual interactive environment. For example, when first learning to drive an automobile, the typical novice driver practices on slow streets or back roads with little traffic. These situations are less demanding than those that the driver will eventually face, yet they help build prerequisite skills incrementally.

In contrast, planning requires advance storage of all potentially pertinent information, making much more severe demands on limited short term memory. It may be that reactive computer systems can efficiently maintain a large short term store for planning of detailed, predictable responses. For humans, however, planning generates first approximations that must be refined through experience. Practice is more robust in that the reliability of its information is known through contact with the environment; planning must take larger portions on faith, making its output less reliably predictable.

A final word about learning is that automatic processing does not perform any.(3) This follows from the fact that automatic processing does not use short term store. If no temporary trace of automatic reactions is maintained, then no new information can be saved as a result of the automatic interaction; the automatic reactions are simply replays of old situation responses.

3. CONTROLLED AND AUTOMATIC COMPUTING

Figure 1 illustrates data flow in our controlled-automatic computing architecture. The controlled partition is in the top half of the figure, the automatic partition in the bottom. This system monitors the surrounding environment through a collection of sensors; the sensors may be polled, and at least some are capable of interrupting ongoing processing by the usual hardware means. The system manipulates the environment through a collection of effectors. Input and output may include terminal based, ASCII data, but will also include physical sense measurements and motor actions in appropriate embedded systems.

<INSERT FIGURE 1 ABOUT HERE.>

-- --

3.1 The automatic partition

The automatic partition is a programmable extension of the combinational digital logic circuitry connecting the computing system to the sensors and effectors. Like combinational circuitry and unlike sequential circuitry, the automatic partition *lacks persistent memory.* By this we mean that the partition does not store information about its previous state configurations.[1] The state of automatic data represents the current state of the sensors, combined with the state of enabling and disabling inputs from the controlled partition. The *decoding* processes of Figure 1 serve as an extension to hardware interrupt recognition and detection. In a conventional computer system, incoming interrupt lines often enjoy a one-to-one correspondence with important conditions that 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 software must take on part of the job of interrupt generation, driving automatic reactions and informing the controlled partition.

Along with lack of persistent memory, key features of the automatic partition are *O(1) complexity* for all code segments, *non-iterative composition* of code segments, and the use of *priorities* to accelerate critical response processing.

O(1) time complexity: Time and memory requirements for code in the automatic partition never grow beyond a predetermined constant limit. The code generator will impose these limits. Verifiable O(1) code is *non-iterative.* That is, it is code that 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 are equivalent to non-iterative code. By tracing possible execution paths, the code generator can calculate worst-case time bounds for non-iterative code. The limitations inherent in non-iterative code restrict the range of size and complexity across which input data to the code may vary.

The code generator must disallow iterative composition of functions. Data flow in the automatic partition takes the form of an acyclic directed graph. Unrestricted cycles would create indeterminacies

in execution time bounds. In Figure 1, cycles in the data flow may occur where information is passed through the controlled partition, but such data flow is not bound by constant time. Data flow within the automatic partition does not loop.

One inherent feedback loop connects to the automatic partition, mediated by the external world. It goes from the effectors through the environment to the sensors. This cycle conveys the effect of external system actions back to the system.

O(1) space complexity: By lack of persistent memory, we mean that the automatic partition does not buffer information. The partition does not require the services of dynamic storage allocation; allocation is static, as with a FORTRAN program. Each data flow path has the capacity to store exactly one datum of the type that flows along that path.[2] Where data are combined by intersecting paths, exactly one composite datum can be stored. Consequently the state of the automatic partition represents the composite present state of the input sensors and inputs from the controlled partition.

Automatic priorities: Until now we have not addressed the issue of sharing central processor time among enabled processes in the automatic partition. If a ready process must wait for computing resources, then its worst-case response time is the sum of its inherent worst-case response time and the worst-case sum of time it must wait for other processes to release resources that it needs.

The design of PRIOPS is based on an underlying uniprocessor machine, although work on multiprocessor production systems[7,17] might be adaptable to PRIOPS' needs. Given the difficulties in sharing a single processor among multiple, time-constrained tasks, *preemptive priorities* determine the order in which automatic tasks get the processor. Critical tasks and automatic tasks with quick response requirements must share the highest priority levels. Worst case response time for a process of a given priority is the sum of the inherent process time plus the times for all other processes of equal or greater priority plus context switching time, over some encompassing time period in which all of these processes may run. For example, suppose a process *P* is triggered once per second and can produce its response in one millisecond. However, *P's* low priority may force it to wait up to 100 milliseconds

every second for all higher and equal priority processes to complete. Then *P's* worst case response time is 101 milliseconds. Any time spent servicing hardware interrupts and direct memory access transfers counts as high priority processing. The danger of losing low priority responses is not a weakness in PRIOPS, but is rather a weakness of processor sharing. Priorities allow an embedded system designer to identify the processes that must be guaranteed processor availability. PRIOPS is not unique in applying preemptive scheduling priorities to a collection of time-constrained tasks; it *is* unique, however, in applying preemptive priorities to Rete matching steps.

Varieties of automatic code: There are three varieties of non-iterative code: *reactive, record* and *predictive. Reactive code* processes information in a path from sensors to effectors. The constant-time bound applies to this complete path. The purpose of reactive code is the generation of time constrained reactions to environmental phenomena. Since the time constraint applies from the instant that the sensation arrives until the instant that the system reacts, we refer to this as an *instantaneous real-time* requirement. This is the variety of automatic code with the tightest response time requirements.

Since the time requirement on reactive code is not averaged across multiple incoming events, there is no need for dynamically varying buffers in a reactive processing sequence. If an incoming datum were to be processed so slowly that a second incoming datum would demand processing resources before the first released them, then the reaction speed would be insufficient. 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 corresponds to the memory-less character of human automatic processing.

Record code conveys data, with possible transformations, from the sensors to dynamic buffers in the controlled partition. The leftmost arrow in the automatic partition of Figure 1 illustrates the location of record code. Timing requirements for this processing are simpler than for reactive processing. The instantaneous real-time constraint applies only to capture of incoming information in a buffer. The buffer smooths variations in the speed of the incoming information flow. In order to avoid buffer

overflow, the control process draining the buffer 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.

Predictive code transmits predetermined actions from controlled partition buffers to effector outputs. These actions are fully predictable at compile time; unlike reactive code, its is not conditioned directly by sensory input. Transferral of output information may be subject to interval timer-based triggering. Like the input portion of record code, the output portion of predictive code is bounded by constant-time limits. The rightmost arrow in the automatic partition of Figure 1 illustrates the location of predictive code. The control process that supplies the action buffer may need to meet an average real-time requirement to prevent the predictive output code from draining the buffer.

3.2 The controlled partition

The controlled partition performs the higher order cognitive tasks of the system. Like controlled processing in humans, the activities of this partition are flexible and powerful; they deal with novel or ambiguous tasks. Search-based, goal and data driven inference can drive controlled partition activities. Problem solving, planning, and learning occur in the controlled domain. Because controlled processing deals with the unknown, and because it requires use of memory and other resources to dynamically varying degrees, we cannot determine a priori constant time and space bounds for control processes.

All persistent storage resides in the controlled partition, including record and predictive code buffers. With the automatic partition acting as a complex device driver for controlled processing, these interface buffers serve as short term sensorimotor storage. Any garbage collection associated with dynamically allocated memory occurs as a controlled partition operation; the automatic partition does not incur the execution overhead of dynamic storage reclamation.

In a planning or learning system, the controlled partition may generate some of the automatic code. Planning can design rough automatic reactions that are refined through practice.

4. A PRIORITIZED PRODUCTION SYSTEM

4.1 A programming notation

Forward-chaining production systems have a stimulus-response organization. They are 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. Frame systems share these characteristics, and also permit procedural attachments of arbitrary computational complexity. The computational complexity of first order logic as a representation language are well known. In contrast, individual production instantiations 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.[3]

Unfortunately, the algorithms implementing the run-time environments for production systems do not guarantee constant-time bounded responsiveness. The Rete pattern matching algorithm[(4,5)] 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.

PRIOPS augments OPS5 notation[(2,6)] with novel, time-constrained semantics.

4.2 A two-tiered example

Figure 2 shows the data flow for a system that monitors temperature sensors. One automatic behavior reacts to dangerously high readings by triggering effector motions - a wheel - away from the heat source, in constant-bound time. An alternative low priority behavior records abnormally high temperature readings into a controlled partition buffer. (Not all high readings are immediately dangerous.) From the controlled partition, abnormal temperatures trigger the creation of a goal to inspect the hot areas. This goal in turn directs effectors to advance toward the hot area - unless emergency

motion (the automatic behavior) is already in progress. The controlled partition establishes temperature thresholds for both partitions at program initialization time.

<INSERT FIGURE 2 ABOUT HERE.>

Important problems within this example include constant-time constrained matching and reaction for automatic productions, and synchronization of efforts between the two partitions. First we introduce the PRIOPS notation and type declaration preliminaries. Next we examine Rete matching as applied in the controlled partition; we choose controlled matching first because it is closer to standard Rete. Then we examine preemptive matching for the automatic partition.

4.3 PRIOPS preliminaries

The syntax and much of the semantics of PRIOPS derive from OPS5.[2,6] Productions represent long term store, working memory elements represent short term store. Inference is a match and act cycle, where one of a set of instantiated productions (the *conflict set* - productions whose left hand side conditions are met by working memory elements) is fired, causing execution of its right hand side actions. These actions may include changes to working memory, and hence to the conflict set. After a fired rule has completed, the cycle begins again. PRIOPS defers and may cancel low priority portions of this match and act cycle. The PRIOPS compiler and run-time support functions are written in C.

Unlike OPS5, PRIOPS applies strong data typing to its working memory element fields. OPS5 is LISP-like in allowing any atomic type to occupy a memory element field. Strong typing eliminates run-time type checking overhead; type checking occurs at compile-time. A future version of PRIOPS could make compile-time type declarations optional; the run-time type tests can be made constant-time bounded, but certain type incompatibilities would not be found until run-time for untyped fields.

Listing 1 shows working memory type declarations in PRIOPS. A memory element field type may be one of *integer, float, symbol* or a *user defined set* type. In Listing 1 we declare set *all-sensors* with a

universe of *t1, t2, t3, t4, tremote1,* and *tremote2.* These six temperature sensors provide the input which drives our example. PRIOPS represents user defined sets as bit maps. The PRIOPS compiler translates a set declaration into a word and bit position mapping for all members of the universe. The size of the universe, and therefore the number of words in a set representation, are known to the compiler. The execution times to perform set operations are available at compile time; primitive machine instructions support these operations.

<INSERT LISTING 1 ABOUT HERE.>

Following the set declaration are *structure* declarations for structured types *limit* and *sensor.* PRIOPS structure declarations take the place of OPS5 *literalize* statements. Each PRIOPS working memory element is an object of one of these structured types, and each field is one of the atomic types already discussed. Listing 1 shows type *limit* containing three fields, *limit-type* (a symbol which determines the use of the limit), *target* (a set of sensors to which this limit applies), and *limit-value* (a floating point threshold value which is the actual limit). Likewise type *sensor* contains three fields, *sensor-id* (a symbolic name for the sensor), *reading* (a floating point measurement), and *direction* (floating point direction of the sensor from the vehicle center, in degrees). Like Pascal records and C structures, PRIOPS computes field address offsets at compile time; unlike OPS5, a field name may not be used for a working memory element type that does not declare that field,[4] and fields are strongly typed. PRIOPS supports typed vector fields similar to singly dimensioned Pascal and C arrays; the present example does not use this capability. The production examples in upcoming sections also use structured types *goal, alarm-move, alarm-record,* and *action.* Listing 1 does not show these structure declarations because they are very similar to the two already discussed; the field types can be determined from usage.

4.4 PRIOPS controlled productions

Listing 2 shows the three controlled productions that correspond to the controlled partition activity of Figure 2. The first production, *attend-to-hitemp,* responds to temperature sensor readings that exceed some *hi-interest* limit. Readers familiar with OPS5 will notice an unexpected *-10* after the production name. This number is a production priority. Priorities range from -128 to 127; a priority less than or equal to 0 places the production into the controlled partition, a priority greater than 0 places the production into the automatic partition.

<INSERT LISTING 2 ABOUT HERE.>

A temperature sensor interrupt handler (code not shown) generates the working memory element matched in the *sensor* test of *attend-to-hitemp.* The complete sensor test, which matches a sensor memory element, is known as a *condition element.* PRIOPS provides mechanisms for C or assembly language device drivers to assert, modify, and retract working memory elements. Normally a sensor input driver will modify its previous reading by removing an outdated working memory element and asserting a new one. The bracketed <*longtemp-id*> and <*hi-reading*> symbols in the sensor test are variables that are bound to the value of the respective fields the first time they appear. Attend-to-hitemp tests a *hi-interest limit* in working memory to see whether the sensor reading equals or exceeds the limit value, and whether the identified sensor is in the set of target sensors for this limit. The >= test here reads *is a superset.* Note that variable <*longtemp-id*> is bound to the *^sensor-id* field value of the sensor memory element, a field which we know from Listing 1 is a *symbol.* The limit test of attend-to-hitemp, however, compares a set containing this variable to the limit's *^target* field value, a value of type *set all-sensors.*

PRIOPS allows symbol variables to appear in user defined set fields in controlled productions. The value of the variable is matched at run-time against the symbols supplied as the universe of the set type at compile-time. Comparison of constant or variable set objects to other set objects is valid in automatic

productions, but this *coercion* of a symbol variable to a component of a set variable is not allowed in automatic productions because of run-time overhead.

When attend-to-hitemp succeeds in finding a targeted sensor reading that exceeds a high interest threshold, the production makes a *goal* type working memory element that identifies the target. The creation of the goal triggers any productions designed to work on that goal, including *move-to-inspect-target.* The latter production takes the target of the goal and resolves its *direction* in space by consulting the sensor memory element. Move-to-inspect-target will not fire, however, if an emergency move, represented by an *alarm-move* element, is under way. If no such emergency is in progress, the production asserts an *action* element to effect the movement.

Finally, *report-alarm* reports alarms generated in the automatic partition to the user console. This production is not in the automatic partition because it is not part of a constant-time bound reaction to the alarm. We have given it a priority of 0 because it acts as a garbage collector for alarm-record elements, deleting them after reporting the alarm. With a priority of 0 it will execute (when satisfied) before all lower priority controlled productions. Explicit memory element garbage collection belongs most appropriately within priority 0 productions; garbage is collected before controlled productions make further dynamic storage requests.

Figure 3 displays the Rete matching network that corresponds to the left hand side tests in the three controlled productions. The round nodes represent tests that apply to a single memory element; these tests compare element field values against constants, or compare several fields within a single working memory element. We refer to them as *pre-join nodes.* Figure 3 shows the *limit ^limit-type = hi-interest* constant test from attend-to-hitemp, and the *goal ^action = inspect* constant test from move-to-inspect-target. When a PRIOPS production or device driver asserts or retracts a working memory element, a reference to the element propagates down the path for its type. A limit element, for example, advances to the pre-join ^limit-type test node of Figure 3. Other production condition element tests which test limit objects differently, would cause branching in this test path. A production which tests for *limit*

^limit-type = hi-danger would generate an alternative limit path which branches before the hi-interest path. Condition elements generate matching paths that are shared to the point where the condition elements themselves differ; identical condition element prefixes generate one path, which diverges at their first point of difference.

<INSERT FIGURE 3 ABOUT HERE.>

Memory element assertions and retractions propagate identically through the pre-join paths. In Figure 3, no pre-join testing of sensor elements occurs because none appear in the three controlled productions.

At the end of a pre-join path these memory element *tokens* are stored in an unbounded memory buffer. Following the first level of memory, the first level *join nodes* perform inter-condition element tests. The threshold limit and target tests of attend-to-hitemp appear in the right elliptical join node of Figure 3. Token pairs that satisfy these inter-element tests are joined and propagated forward through the network. Attend-to-hitemp contains only two condition elements, so only one join occurs. Production move-to-inspect target contains three condition elements, so two consecutive joins occur. The second join in that path is an alternative join called a *not* node, because it succeeds only for tokens arriving at its left input that cannot be joined to tokens on its right input using the inter-element tests. In our example there are no inter-element tests, so the mere presence of an alarm-move token is enough to halt token propagation. Except for not nodes, successfully matched assertions of tokens results in propagating assertions of joined tokens; successfully matched retractions of tokens results in propagating retractions of joined tokens. Successfully matched not tests can result in the retraction of left input tokens that were previously asserted for asserted right input tokens, and in the assertion of left input tokens that were previously inhibited for retracted right input tokens.

Tokens successfully tested and joined to the end of a path contribute production instantiations to the

conflict set. At the bottom of Figure 3 are conflict set entry points for attend-to-hitemp, move-to-inspect-target, and report-alarm test paths. PRIOPS augments OPS5 usual methods for determining which member of the conflict set to fire (the *conflict resolution strategy).* OPS5 uses *recency* of participating memory element assertions and *specificity* of condition element tests to select a production instantiation to fire. For controlled productions, PRIOPS uses *production priority* as the main conflict resolution criterion, and uses recency and specificity to break ties.

As Haley[8] has pointed out, a compiler can compute weak worst-case execution times for pre-join portions of a Rete net at compile-time. The most pessimistic estimate is the sum of the time to execute all pre-join tests for a specific working memory element class. A better estimate is available when paths are mutually exclusive in the elements they will match. PRIOPS can detect such exclusions by scanning the successors to a node in a pre-join chain. It looks for distinct constant values, non-overlapping numeric ranges, or non-intersecting set values. The goal of PRIOPS is acceptable as well as constant time bounds.

Haley has also shown that the time to compute joins dominates the matching time of Rete. The compiler cannot determine worst-case join times because traditional Rete does not restrict the size of memory nodes. In Figure 3, for example, many hi-interest limit tokens might propagate to the first limit memory node. A sensor token entering the left side of the join node matches against all limit tokens stored to the right. Join time is therefore a function of both the sizes of contributing memories, and the tests performed within the join node. Memory sizes are unknowable at compile-time. We see here at the detailed matching level that persistent memory contributes to execution time indeterminacy. PRIOPS eliminates unbounded memory nodes from the Rete net in the automatic partition.

4.5 PRIOPS automatic productions

Listing 3 shows the three automatic productions that correspond to the automatic partition activity of Figure 2. The three productions implement a reactive path through the automatic partition for response to dangerously high temperature readings; they also coordinate with the controlled partition. Priorities

greater than 0 identify these as automatic productions.

<INSERT LISTING 3 ABOUT HERE.>

Production *react-to-overtemp,* with a priority of 10, recognizes a dangerous temperature condition and responds by initiating several activities. The *^sensor-id* test @ *[t1 t2 t3 t4]* <*target-id*> @ constitutes a PRIOPS *macro.* Values t1, t2, t3, and t4 direct alternative expansions of the macro. When the PRIOPS compiler parses a macro, it replaces the *base production* (react-to-overtemp in this case) with a distinct production for each distinct expansion of the macro. Each production will test the field differently. Expanded production *react-to-overtemp-t1* will test *^sensor-id t1,* production *react-to-overtemp-t2* will test *^sensor-id t2,* and so on. Generation of condition elements that diverge at the point of the macro, results in generation of *mutually exclusive* Rete matching paths that diverge at the point of the macro. Listing 4 shows one possible expansion of base production *react-to-overtemp* from Listing 3.

<INSERT LISTING 4 ABOUT HERE.>

The identifier <*target-id*> within the macro call appears to be a regular PRIOPS variable. In fact, <target-id> here is a *macro tag.* For each expansion of the macro, the selected macro expansion replaces the macro tag wherever the latter appears in the production. For instance, in production react-to-overtemp-t1, constant *t1* replaces all occurrences of <*target-id*>. A macro tag may appear any place within a production that a constant may appear; there are places that constants may appear but variables may not (e.g., symbols within automatic production set fields as discussed in the previous section). React-to-overtemp-t1 matches readings from sensor t1, binding the direction to variable <*way*> and the reading to <*hitemp*>. The limit test checks for type *hi-danger* this time; the *^target* test is a normal superset comparison as in attend-to-hitemp from Listing 2; the set *[t1 t2 t3 t4]* is a constant. When the

t1 sensor reading exceeds the threshold, react-to-overtemp-t1 fires. The right hand side makes an *alarm-record* for production *report-alarm* in the controlled partition, makes an *alarm-move* to inhibit *move-to-inspect-target* and any other lower priority move command, computes the direction opposite to the heat source, and issues a *move action* to initiate effector action.

Retract-overtemp triggers when a temperature emergency condition terminates. The *alarm-move* test detects the emergency, and the remaining tests determine that the high temperature problem no longer exists. The right hand side actions of the production are not important to this discussion.

Start-moving is the third automatic production. It takes an *action* element and initiates motion by calling a C language device driver *wheel_driver.* Whereas input interrupt handlers normally communicate to PRIOPS by modifying sensor working memory elements, PRIOPS productions trigger output drivers by calling them directly. The wheel driver initiates the action, but only if the command <*urgency*> exceeds the <urgency> of the most recent action command to the wheels that is still in effect. Assume that if no wheel_driver call occurs within 30 seconds, the driver times out and decelerates the wheels. Assume further that temperature readings arrive many times a second, so the wheel time-out is strictly a default behavior.

In order to fully appreciate the interaction of these productions with each other and with the controlled productions, examine the matching network of Figure 4. Only the t1 expansions appear in the figure; alternative paths for the t2, t3, and t4 expanded productions parallel the t1 paths. Note that each node is tagged with the priority of the production that generated it. In cases where several condition elements contribute to a single node, the node takes on the priority of the highest priority contributing production. For example, the *^sensor-id t1* constant test node in Figure 4 is part of the matching for the first condition element of react-to-overtemp-t1 (priority 10), and for the third condition element of retract-overtemp-t1 (priority 1), so the shared node receives a priority of 10.

<INSERT FIGURE 4 ABOUT HERE.>

PRIOPS uses priorities to defer portions of Rete matching. PRIOPS maintains a priority queue of matching tasks. When a working memory change occurs, the matching action is queued in the appropriate queue; there is one queue for each automatic level, and a single queue for the controlled partition. Controlled matching only proceeds when no automatic matching or actions are occurring, and controlled matching does not use priority information until conflict resolution, so ignore the controlled partition for now.

Automatic matching uses priority information at each step; matching tasks are queued on a per node basis. For example, assume that react-to-overtemp-t1's right hand side has just made an alarm-move element and an action element. Figure 4 shows that the alarm-move matching commences with priority 1; it will remain in its queue while the priority 100 action matching proceeds. After the matching at a single node succeeds, match tasks for all successor nodes enter appropriate queues. Successor priorities will always be less than or equal to the current priority, because shared condition element prefixes take on the priority of the greatest contributing production; when the paths diverge, some priorities may diminish because the greatest contributing production priority is less. An example occurs at the sensor *register* node; the right successor, contributed by react-to-overtemp-t1, has priority 10; the left successor, contributed by retract-overtemp-t1, has priority 1.

Unlike OPS5, matching is not intermixed with the right hand side actions of PRIOPS productions. OPS5 performs matching immediately for each right hand side change to working memory.[2, p. 230] In PRIOPS, all assertions to and retractions from working memory are committed, and then matching proceeds; this allows the highest priority matching to proceed first, regardless of the action order within the production right hand side. Incoming interrupt driven memory changes will interrupt both controlled activity and lower priority automatic activity. There is minor synchronization overhead associated with handling critical section problems; we will not discuss these details here.

Besides the use of priority matching queue, the second major distinction in automatic partition Rete is the appearance of the *register* nodes of Figure 4 in place of the *memory* nodes of Figure 3. Unlike the

unbounded controlled matching memory nodes, each register can hold only one token. Assertion of any new token at an automatic node causes automatic retraction of all old information along descendent paths. New information always replaces old.

The restriction of unit register 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 registers that reflect the current state of the environment. This restriction on size accords well with observations on human automatic processing. 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.[(10)] 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 register nodes are of such limited capacity, it is necessary for the compiler to generate distinct register 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. PRIOPS matching optimizes token propagation in the automatic partition by replacing the combined *retract old* and *assert new* steps needed in controlled matching with *modify* token propagation. The modify retracts the previous token and asserts the new. When the new token passes node tests, the modify propagates forward; when the new token fails, a retraction of the old token propagates instead.

Another consequence of unit register size is that only one instantiation of a given automatic production can be ready at a point in time; in OPS5 and in the PRIOPS controlled partition, a single production's condition elements may be satisfied by different combinations of memory elements, so one

production may be instantiated several different ways at one time. Since the compiler knows the number of automatic productions and automatic matching nodes, it also knows the upper limit on the size of the automatic conflict set and automatic priority queues respectively. All automatic activity is of O(1) complexity.

Because of the one-to-one correspondence between distinct sensors and paths through automatic Rete, PRIOPS supplies the macro capability to make the generation of these matching paths less painful. Thus while *react-to-overtemp* is written as one production, macro expansion allows it to generate diverging matching paths for each sensor whose state must be stored. Sharing a single register among multiple sensors would cause the most recent sensor reading to overwrite readings for other sensors. Note that the executable code for macro generated matching *is shared.* The nodes are distinct, but parallel expanded nodes point to identical executable test code.

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. When no automatic productions are instantiated, matching of controlled productions ensues.

Returning to Figure 4, assume that some unrelated controlled matching is taking place when a dangerously high temperature reading comes from sensor t1. Temperature limits were asserted at initialization time. The sensor identification succeeds and the sensor reading is saved in its register; the priority 1 sensor join matching waits while the priority 10 limit join testing proceeds. The test passes and a priority 10 production - react-to-overtemp-t1 - enters the automatic conflict set. No other priority 10 or greater activity is queued, so the production fires, making three working memory elements. The action, at priority 100, is the most salient, so its matching proceeds; note that the priority 1 sensor

reading remains queued. Priority 100 matching causes the most urgent action - the wheel movement initiation - to proceed without regard to lower priority pending activity. The path from the t1 sensor test through the start-moving firing is acyclic, and the matching time is constant bounded.

Some of the queued activity may become outdated before it ever has an opportunity to execute. The queued priority 1 sensor task may remain queued while another t1 sensor reading occurs. Because of this possibility, only one matching task can be queued at an automatic node at one time. A newer task always replaces an older, queued task, because at that point the older information has become outdated. Outdated tasks within the controlled partition propagate fully because an unlimited number of tasks can be queued at a single node; any retractions will cancel outdated assertions before controlled conflict resolution takes place.

The theory of human automatic processing provides a model for the type of activity that can be automated: repetitive, predictable, usually appropriate reactions. PRIOPS provides a collection of mechanisms for supporting such activities in an embedded software system in constant bounded time. Implementing practical control applications using this architecture is primarily a software engineering problem.

5. RELATED WORK AND FUTURE RESEARCH

Related work on partitioning production memory has been performed in the PAMELA system,[1] 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 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 assertion, retraction, 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 cancelled. Automatic rules respond primarily to sensory data; once triggered, these execute to completion in accordance with the principles of human automatic processing. 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.

An additional area that needs study is the subject of planning and learning in the controlled partition. Production *chunking* is one learning mechanism that appears appropriate to PRIOPS.[9] Chunking generates specialized *chunk productions* for using specific knowledge acquired during problem solving; the availability of the chunk allows the *SOAR* system to avoid repeating a problem solving episode by using knowledge compiled during the original problem solving episode. PRIOPS might generate specialized automatic chunks from controlled problem solving episodes. We plan to build learning machinery into a future version of PRIOPS to explore this possibility.

6. 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.

NOTES

1. *Persistent memory* is analogous to *long-term memory* in humans; memories about details of an episode remain after the episode is complete. This contrasts with *short-term memory,* which provides transient buffering. Persistent memory will be discussed more thoroughly in the section on *O(1) space complexity.*

2. O(1) space complexity does not require that we restrict storage capacity to exactly one datum, only to a constant bound amount. We have chosen a limit of one in order to force the automatic partition to represent the current state of the sensors, without history.

3. For production systems with learning mechanisms, the definition of *compile-time* is ambiguous. Processing which learns by building and compiling productions may execute at *run-time.* Compile-time for learned productions coincides with run-time for the learning mechanism and other parts of the production system program. The PRIOPS compiler that compiles learned productions at run-time is *not* an automatic process (i.e., it is not O(1)); this compiler *can* compile automatic productions and analyze their timing characteristics as it compiles.

4. Two valid OPS5 type declarations are *(literalize cat purr)* and *(literalize mouse squeak).* The first defines type *cat* with field *purr,* the second, type *mouse* with field *squeak.* After declaring these working memory types, the OPS5 programmer is free to manipulate the *squeak* field of *cat* and the *purr* field of *mouse,* even though these field names were not declared for these types.

--

--

REFERENCES

1. Barachini, Franz 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.

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

3. Fisk, Arthur D. 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.

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

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

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

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

8. Haley, Paul V., "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.

9. Laird, John E., Paul S. Rosenbloom and Allen Newell, "Chunking in SOAR: The Anatomy of a General Learning Mechanism." Memo CMU-CS-85-154, Carnegie-Mellon University, September, 1985.

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

11. Parson, Dale E. and Glenn D. Blank, "Constant-time pattern matching for real-time production systems," *Proceedings of Applications of Artificial Intelligence VII,* Vol. 1095, Part 2, ed. Mohan M Trivedi. Bellingham, Washington: Society of Photo-Optical Instrumentation Engineers, 1989, p. 971-

982.

12. Schneider, Walter and Arthur D. Fisk, "Degree of Consistent Training and the Development of Automatic Processing," Human Attention Research Laboratory Report No. 8005, University of Illinois, February, 1980.

13. Schneider, Walter 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.

14. Schneider, Walter 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.

15. Shiffrin, Richard M. 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.

16. Shiffrin, Richard M. 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.

17. Tambe, Milind, Dirk Kalp, Anoop Gupta, Charles Forgy, Brian Milnes and Allen Newell, "SOAR/PSM-E: Investigating Match Parallelism in a Learning Production System." *ACM SIGPLAN Notices* 23(9) (September, 1988), p. 146-160.

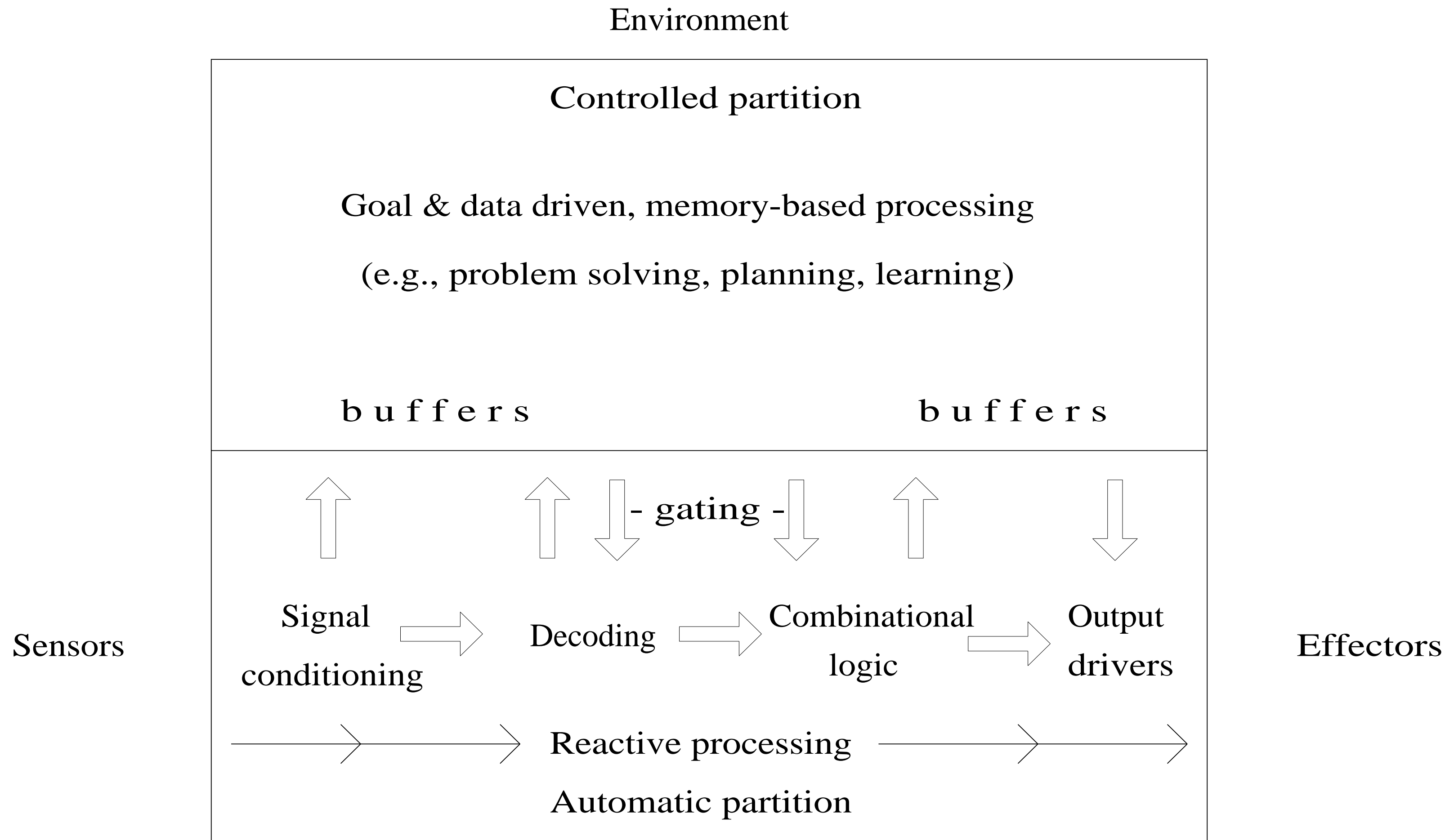

Figure 1 - The two-tiered, controlled-automatic architecture

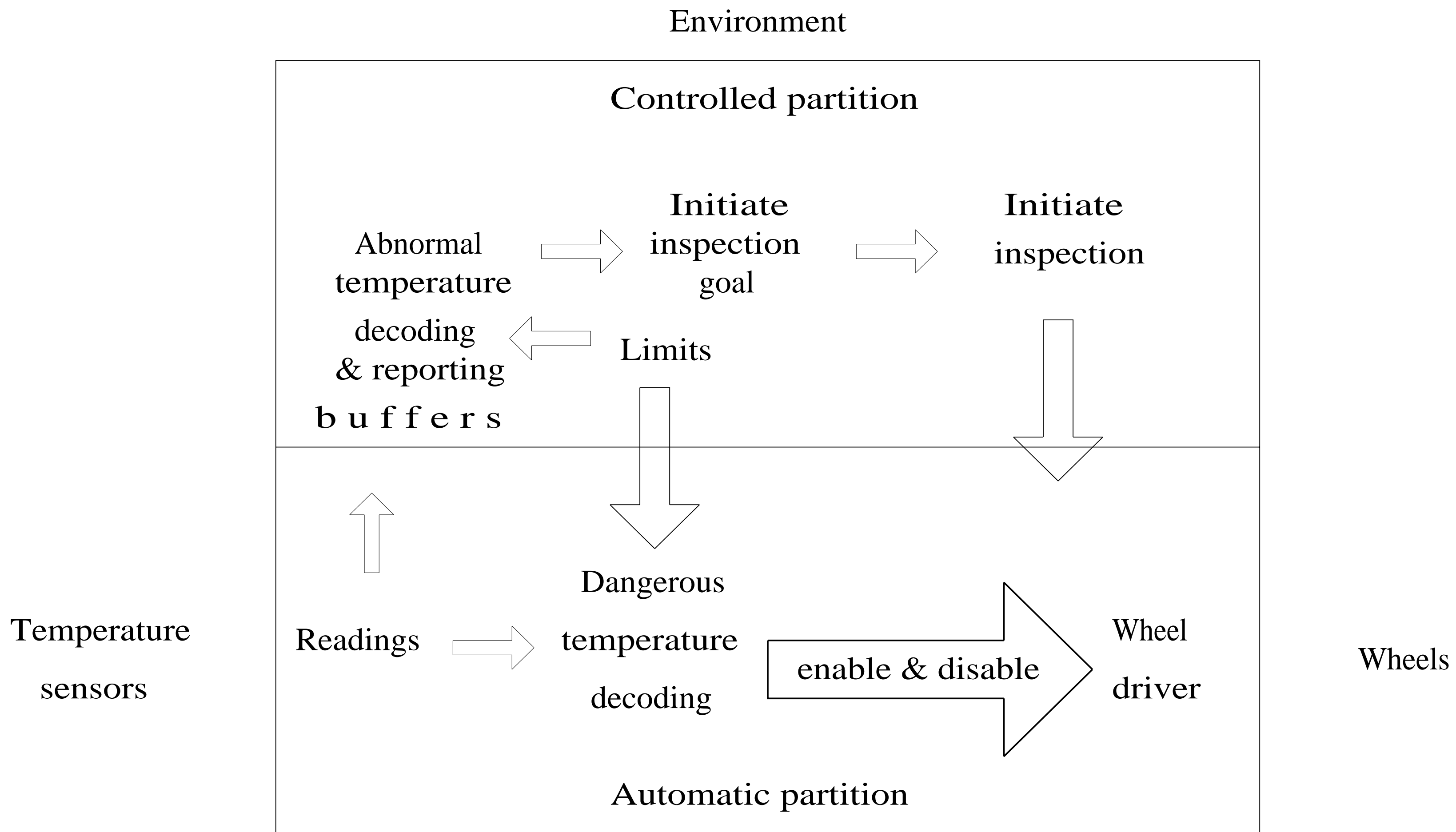

Figure 2 - A temperature sensor driven example

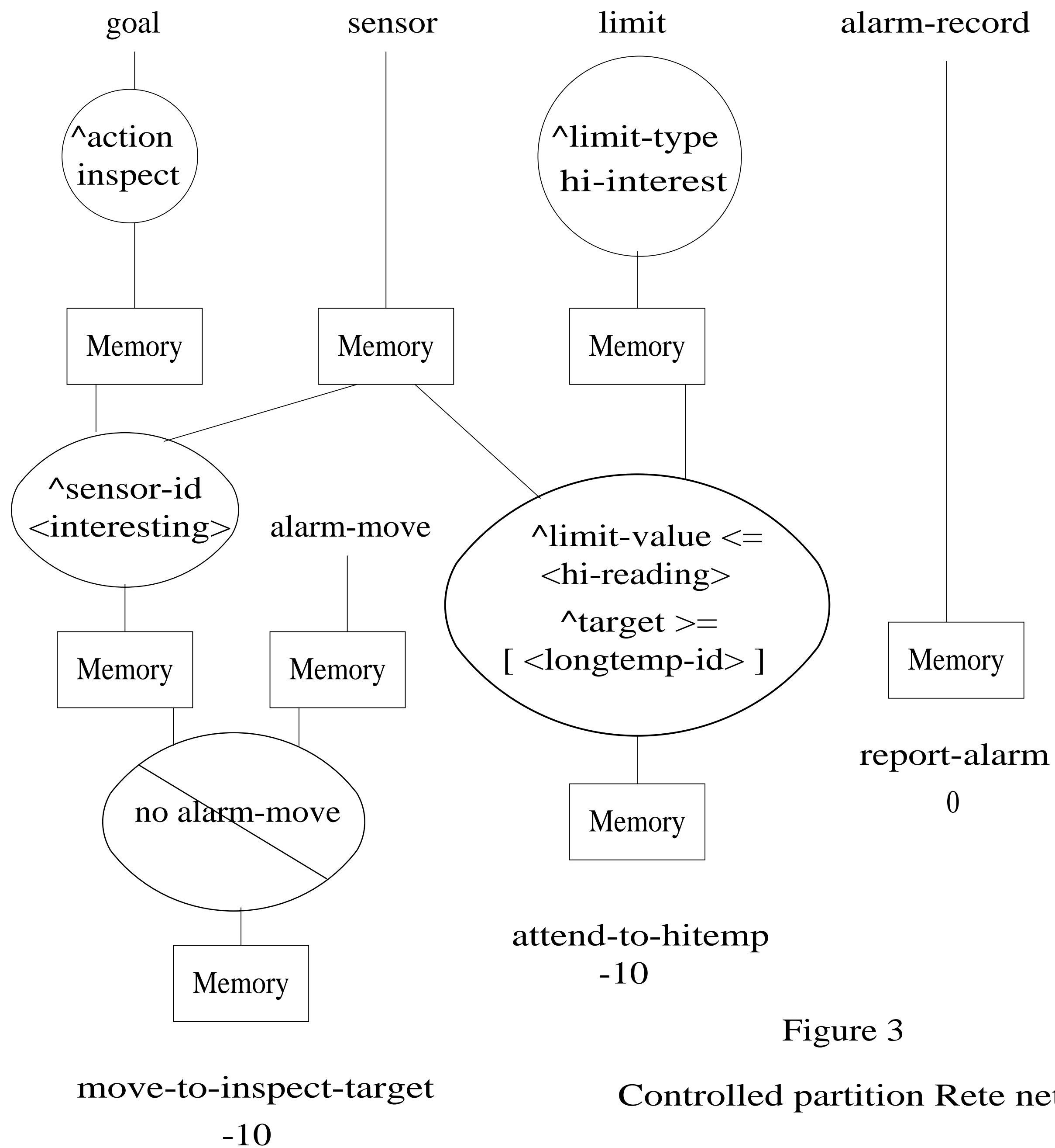

Figure 3

Controlled partition Rete net

alarm-move
sensor
limit
action

1
^alarm-type
hi-temperature

t2
t3
t4
(additional
t2 t3 t4 paths)

10
^sensor-id
t1

10
^limit-type
hi-danger

100
^action-type
move

1
^alarmed-sensor
t1

10
^target >=
[t1 t2
t3 t4]

100
^action-target
wheels

10
Register

10
Register

Register
100

1
Register

start-moving
100

10
^limit-value
<= <hitemp>

1
no tests

10
Register

1
Register

1
^reading >=
<hi-danger-limit>

react-to-overtemp-t1
10

1
Register

retract-overtemp-t1
1

Figure 4
Automatic partition Rete net

```
(set all-sensors                ; set of all sensors
        t1 t2 t3 t4             ; short range temperature sensors
        tremote1 tremote2       ; long range, infrared temperature sensors
)

(structure limit                ; limits on sensor readings
        symbol limit-type       ; class of limit determines response
        set all-sensors target  ; sensor set targeted by this limit
        float limit-value       ; threshold
)

(structure sensor               ; sensor reading representation
        symbol sensor-id        ; unique sensor identification
        float reading           ; measurement
        float direction         ; direction of sensor from vehicle
                                ; center in degrees, 0 is due north
)
```

Listing 1 - Working memory type declarations in PRIOPS

```
(p attend-to-hitemp -10                    ; notice an unusually hi reading
        (sensor ^sensor-id <longtemp-id> ^reading <hi-reading>)
        (limit ^limit-type hi-interest
                ^limit-value <= <hi-reading>
                ^target >= [ <longtemp-id> ] )
                ; target set is SUPERSET of set containing <longtemp-id>
                ; [ f1 f2 ] reads: "a set containing f1 and f2"
        -->
        (make goal ^action inspect ^target <longtemp-id>)
)

(p move-to-inspect-target -10           ; move to find cause of hi reading
        (goal ^action inspect ^target <interesting>)
        (sensor ^sensor-id <interesting> ^direction <way>)
        - (alarm-move)
        -->
        (make action ^action-type move ^action-target wheels
                ^direction <way> ^speed slow ^urgency -10)
)

(p report-alarm 0                    ; report alarms from automatic partition
        {(alarm-record ^alarm-type <type> ^alarmed-sensor <sensor>
                ^limit <limit> ^reading <reading>) <alarmrec>}
        -->
        (write Alarm type <type> on sensor <sensor> ",")
        (write limit = <limit> "," reading = <reading> "." (crlf))
        (remove <alarmrec>)
)
```

Listing 2 - Controlled partition productions

```
(p react-to-overtemp 10          ; trigger emergency response, record
    (sensor ^sensor-id @ [t1 t2 t3 t4] <target-id> @
        ^direction <way> ^reading <hitemp>)
    (limit ^limit-type hi-danger
        ^target >= [t1 t2 t3 t4]
        ^limit-value {<hi-danger-limit> <= <hitemp>})
    -->
    (make alarm-record ^alarm-type hi-temperature
        ^alarmed-sensor <target-id>
        ^limit <hi-danger-limit> ^reading <hitemp>)
    (make alarm-move ^alarm-type hi-temperature
        ^alarmed-sensor <target-id>)
    (bind <newway> (compute (<way> + 180) % 360))
    (make action ^action-type move ^action-target wheels
        ^direction <newway> ^speed fast ^urgency 10)
)

(p retract-overtemp 1      ; retract emergency status
    (alarm-move ^alarm-type hi-temperature
        ^alarmed-sensor @ [t1 t2 t3 t4] <target-id> @)
    (limit ^limit-type hi-danger
        ^target >= [t1 t2 t3 t4]
        ^limit-value <hi-danger-limit>)
    -(sensor ^sensor-id <target-id>
        ^reading >= <hi-danger-limit>)
    -->
    ... end of emergency actions - not shown ...
)

(p start-moving 100      ; drive output
    (action ^action-type move ^action-target wheels
        ^direction <where> ^speed <velocity> ^urgency <urgency>)
    -->
    (call wheel_driver <where> <velocity> <urgency>)
)
```

Listing 3 - Automatic partition productions

```
(p react-to-overtemp 10              ; base production with macro
        (sensor ^sensor-id @ [t1 t2 t3 t4] <target-id> @
                ^direction <way> ^reading <hitemp>)
        (limit ^limit-type hi-danger
                ^target >= [t1 t2 t3 t4]
                ^limit-value {<hi-danger-limit> <= <hitemp>})
        -->
        (make alarm-record ^alarm-type hi-temperature
                ^alarmed-sensor <target-id>
                ^limit <hi-danger-limit> ^reading <hitemp>)
        (make alarm-move ^alarm-type hi-temperature
                ^alarmed-sensor <target-id>)
        (bind <newway> (compute (<way> + 180) % 360))
        (make action ^action-type move ^action-target wheels
                ^direction <newway> ^speed fast ^urgency 10)
)

(p react-to-overtemp-t1 10           ; an expansion of react-to-overtemp
        (sensor ^sensor-id t1        ; t1 substitution
                ^direction <way> ^reading <hitemp>)
        (limit ^limit-type hi-danger
                ^target >= [t1 t2 t3 t4]
                ^limit-value {<hi-danger-limit> <= <hitemp>})
        -->
        (make alarm-record ^alarm-type hi-temperature
                ^alarmed-sensor t1   ; t1 substitution
                ^limit <hi-danger-limit> ^reading <hitemp>)
        (make alarm-move ^alarm-type hi-temperature
                ^alarmed-sensor t1   ; t1 substitution
        (bind <newway> (compute (<way> + 180) % 360))
        (make action ^action-type move ^action-target wheels
                ^direction <newway> ^speed fast ^urgency 10)
)
```

Listing 4 - Macro expansion of react-to-overtemp