PRIOPS: A real-time production system architecture for programming and learning in embedded 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 Pattern Recognition and Artificial Intelligence,* Volume 4, Number 3 (1990), p. 509-526.

PRIOPS: A real-time production system architecture for programming and learning in embedded systems

ABSTRACT

The Prioritized Production System - *PRIOPS* - is an architecture that supports time-constrained, knowledge-based embedded system programming and learning. Inspired by cognitive psychology's theory of *automatic* and *controlled human information processing*, PRIOPS supports a two-tiered processing approach. The *automatic partition* provides for compilation of productions into *constant-time-constrained processes* for reaction to environmental conditions. The notion of a *habit* in humans approximates the concept of automatic processing, trading flexibility and generality for efficiency and predictability in dealing with expected environmental situations. *Explicit priorities* allow critical automatic activities to preempt and defer execution of lower priority processing. An augmented version of the *Rete match algorithm* implements O(1), priority-scheduled automatic matching. PRIOPS' *controlled partition* supports more complex, less predictable activities such as problem solving, planning and learning that apply in novel situations for which automatic reactions do not exist. The PRIOPS notation allows the programmer of knowledge-based embedded systems to work at a more appropriate level of abstraction than is provided by conventional embedded systems programming techniques. This paper explores programming and learning in PRIOPS in the context of a maze traversal program.

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

1. INTRODUCTION

Embedded computing systems do more than consume and produce symbolic text and graphics for direct human interaction. These systems deal with physical phenomena such as visual images, sound, motion, pressure and temperature. Measurements enter the computing systems through transducers and sensors. These systems use effectors and generators, such as robotic arms and lasers, to produce changes in the sensed phenomena. While symbolic information may appear at the input and output ports of these systems, it usually does so alongside more critical signals and actions that constitute direct interactions with external environments.

A typical embedded computing system must react to important environmental conditions in predictable time. The system must recognize an important external situation in limited time, and then respond in limited time. The higher priority work of an embedded system - reading critical sensors, generating effector control signals, sounding alarms - exhibits this stimulus-response organization. Lower priority activities, such as accounting and routine report generation, execute in the background during lulls in environmental interaction. These processes do not have strict time requirements.

Interrupt-driven event interpretation and reaction has traditionally used assembly language programming. Subsequent work applied the improved constructs of procedural(24) and object-oriented(9) languages to the problems of time-constrained programming. Recently work has begun on adapting the notations and architectures of knowledge-based systems for use in real-time embedded applications.(10)

The Prioritized Production System *(PRIOPS)* is an architecture that supports the programming and learning of event-driven, constant-time-constrained, preemptive stimulus-response processes. PRIOPS notation is a derivative of OPS5.(2,5) PRIOPS pattern matching uses an augmented version of OPS5's Rete matcher to select productions for execution.(3,4) In earlier papers we reported on the PRIOPS architecture and matching algorithm,(16,17) relating key aspects of this two-tiered architecture to cognitive psychology's concepts of *controlled* and *automatic* human information processing.(18,19,20) Here we use an example PRIOPS program to explore application of the architecture. The example, which searches a maze for its exit while avoiding dangerous obstacles, presents a method for learning PRIOPS real-time processes.

2. EMBEDDED REAL-TIME COMPUTING

The expression *embedded computing system* identifies a computer that is part of an encompassing piece of machinery or larger physical system. Popular definitions of embedded system do not address questions of time or space locality, so for precision we enumerate properties of an embedded system:

1) The processing system is embedded in a discrete, functioning physical system.

2) The discrete, functioning system is embedded in an environment.

3) Sensors describe the environment to the processing system.

4) Effectors convey the processing system's reactions to the environment.

5) The intersection of the current environmental state and the current processing state is non-empty.

6) Physical localization of the system is important. A restriction on distribution of the embedded system in space and time, the *body concept*, is characterized by the remaining constraints.

7) Intra-system communication speeds for multiple-processor systems are of the same magnitude as processor speeds. The presence of multiple processors does not necessitate internal communication queuing delays.

8) A single system clock/time is accurate for all processors within acceptable error. An embedded system does not require the notion of relativistic, partially ordered time found in distributed systems.(1)

9) The system's physical boundary describes an abstract interface to the environment. The interface is abstract because the sensors do not (normally) exhaustively describe conditions at the boundary. Sensors supply partial information.

Real-time processing in PRIOPS is equivalent to *constant-time-constrained* or *O(1)* processing. The production compiler can determine worst-case execution time for real-time processes.

3. The TWO-TIERED PRIOPS ARCHITECTURE

A PRIOPS programmer partitions the program problem space in two. Problems amenable to solution using recognition and reaction processes with O(1) time and space requirements go in the *automatic partition* (AP). All other problems go in the *controlled partition* (CP).

Cognitive psychology's concepts of automatic and controlled human information processing inspired the PRIOPS architecture.(18,19,20) Automatic productions are like habits. Habits are akin to reflexes or instincts, except that they are learned, through practice. Habits develop in response to repetitive, consistent patterns of interaction with one's environment. Habits trade flexibility for predictability and efficiency. Habits trigger in response to familiar conditions, and produce effective responses to conditions while placing minimal load on the resources of attention and short-term memory. An automatic process may execute without the conscious awareness of the person, and is difficult to avoid or preempt. Human controlled processing, in contrast, is thought- and memory-intensive. Controlled processing is appropriate for novel situations requiring complicated activities such as planning and learning. This processing is far more complex, and therefore less efficient and predictable, than automatic processing. The goal-directed, computation-bound processing typical of artificial intelligence architectures corresponds to controlled processing.

3.1 Data flow in the automatic partition

Figure 1 illustrates data flow in the PRIOPS architecture. Sensors provide information about the environment to the system. The system manipulates the environment through its effectors. *Reactive processing* stimulus-response production chains guarantee real-time responses to external triggers. Reactive productions lie wholly within the AP. Reactive processing flows from sensors through automatic detection to generated actions. There are no unbounded loops in purely automatic data flow, since a loop implies time indeterminacy. The only feedback loop in the AP is an inherent one, mediated by the external world. It goes from effectors through the environment to the sensors. (Internal feedback loops are possible in the CP.)

<INSERT FIGURE 1 ABOUT HERE.>

The AP handles time-critical input-output, recognition and reaction. Being composed primarily of PRIOPS productions, it is in effect a symbol-based interrupt handler and device driver. A small amount of procedural code in a language such as C or Modula-2 may appear for portions of the automatic problem space readily handled by such code. Simple *decoding* activities extract some important environmental information from an individual sensor's data messages. Combining sensed data through *sensor fusion* is necessary for more complex phenomena.

Production systems spend most of their time matching data to conditions (rule left-hand-sides).(4) PRIOPS' augmented Rete matcher assures that automatic match time is constant. Automatic conflict resolution and right-hand-side actions also run in constant time.

Automatic productions show *constant-bound space complexity* as well. Variable length data buffering is a means for lowering responsiveness requirements by storing incoming information until the system has time to attend. PRIOPS assumes a constant-bound responsiveness requirement on each reaction to incoming information, measured from the instant that a stimulus arrives until the instant that a response proceeds. Reactive processing must satisfy *instantaneous real-time* requirements. There is no degradation of responsiveness through arbitrary-length buffering in the AP. Furthermore, incoming information on a sensory channel supersedes any earlier information from that channel. Variable-length buffering of sensory data is again inappropriate, since the AP reacts to the immediate environmental situation.

3.2 The controlled-automatic interface

The automatic productions of Figure 1 trigger not only from sensory input, but also from enabling information advanced from the CP. Where short-term memory of stimuli and strategy information is necessary, the CP performs buffering. The controlled portion of such processing does not guarantee O(1) responsiveness. Buffering at the automatic-controlled interface delays lower-priority controlled computation without losing information. PRIOPS places a constant bound on the length of these buffers, and allocates their space during controlled processing intervals. Automatic processing does not perform memory management, since the complexity of memory management is normally greater than

O(1). Control processes draining the buffers cannot react to a particular buffered datum within a constant time limit, but these processes must drain the buffers at a rate that will keep buffer space from becoming exhausted. Therefore control processes must drain the buffers at the same average rate at which automatic processes load them. Interface control processes must satisfy *average real-time* requirements.

3.3 Explicit production priorities

We have not yet addressed the issue of sharing central processor time among enabled constant-time processes in the AP. (The current PRIOPS implementation is for a uniprocessor machine, although work on multiprocessor production systems(7,21) might be adaptable to PRIOPS' needs.) 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 waits for other processes to release resources that it needs. Given the difficulties in sharing a single processor among multiple, time-constrained tasks, *explicit preemptive priorities* determine the order in which automatic PRIOPS productions get the processor. Priority levels vary from 1 to 127 for automatic productions, and from 0 to -128 for controlled productions. Critical tasks and automatic tasks with quick response requirements 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. 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.

3.4 Controlled partition activities

The CP performs the higher order tasks typical of artificial intelligence architectures. Like controlled processing in humans, the activities of this partition are complex and adaptable, dealing with unexpected or uncertain conditions. Search-based, goal and data driven inference can drive CP 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 long-term information storage resides in the CP.

In a planning or learning system, the CP will generate some of the automatic code. Planning can design rough automatic reactions that are refined through practice. This paper gives one example of a means for generating automatic reactions from the results of controlled search activity.

4. THE MAZE DEMONSTRATION PROGRAM

In the spirit of Pavlov, we shall observe the behavior of a PRIOPS program in a simulated maze. Figure 2 shows an example maze as displayed on a PC monitor. Walls are shaded and tunnels are unshaded; the **EXIT** appears near the upper right corner. Within the maze we shall place two mobile entities: the PRIOPS **PROGRAM** and the **HUMAN** (letters **P** and **H** in Figure 2 denote the respective starting positions of these two entities). The PROGRAM a priori wants to locate and reach the EXIT. The HUMAN, under keyboard control, is dangerous to the PROGRAM. It will destroy the PROGRAM upon contact. So the PROGRAM instinctively avoids the HUMAN.

<INSERT FIGURE 2 (screen shot of a maze) ABOUT HERE.>

Though a toy problem, the maze is rich enough to demonstrate several key ideas of PRIOPS

- Sensor monitoring, here responding to obstacles in the maze.
- Controlled behaviors, notably searching for the EXIT.

- Automatic behaviors, such as fleeing from the HUMAN.
- Garbage collection, at the automatic-controlled interface.
- Learning, here of a successful path to the EXIT.
- Improved behavior, once learning becomes automatic.

The maze, the keyboard-to-maze interface, and the simulated sensors are implemented in C. The PROGRAM itself is a set of PRIOPS productions. The PROGRAM receives sensory input upon beginning execution, upon movement of the PROGRAM, and upon movement of the HUMAN when this movement reaches the PROGRAM's sensors. This input comes from maze C functions. A *sensors* record reports the identity and distance of the nearest obstacle to the *left*, *right*, *up* and *down.* Potential obstacles include a **WALL,** the **HUMAN** opponent, and the sought-after **EXIT.**

4.1 An overview of maze processing

Figure 3 shows data flow for the PROGRAM during maze traversal. All automatic operations trigger on immediate sensory information. These productions act during an emergency to move the PROGRAM within constant-bound time, preempting lower-priority tasks. Automatic maze productions come in three varieties:

1) EXIT detectors. Excited when a sensor detects the EXIT, they preempt all other productions and lunge out of the maze.

2) HUMAN detectors. Triggered when a sensor detects the HUMAN, they flee. The PROGRAM prefers to escape at right-angles to the HUMAN when possible.

3) Learned productions that direct the PROGRAM down the path of escape.

<INSERT FIGURE 3 ABOUT HERE.>

Controlled productions maintain memory of visited maze locations. PRIOPS does not use controlled priorities (0 to -128) for preemption, so in the absence of automatic activity, all controlled matching completes before controlled conflict resolution selects a controlled production instantiation to fire. Controlled maze productions come in five flavors:

1) Initialization: setting up the maze and PROGRAM for execution.

2) Heuristic search: selecting the next move in search of the EXIT.

3) Record-keeping: accumulating declarative knowledge about the maze in working memory.

4) Garbage collection. Garbage can include outdated and redundant information in CP buffers.

5) Learning.

The overall flow of the maze program as seen in Figure 3 typifies the flow and interaction of automatic and controlled processing in PRIOPS. During the search phase of the problem, controlled productions guide the search and collect declarative information. Initial hand-coded automatic productions deal with time-critical situations by changing the immediate relationship of the embedded system to other entities in the environment. Garbage collection at the controlled-automatic interface performs resource reclamation without slowing automatic reactions. Learning takes the results of controlled search and builds them into reactive automatic productions. The following subsections will examine productions from each part of Figure 3, from sensory readings and memory, to search behaviors in the CP, to instinctual behaviors in the AP, to garbage collection at the controlled-automatic interface, to learning in the CP, and finally to resulting learned behaviors in the AP. The final subsection discusses execution results for the maze of Figure 2.

4.2 Sensa, memory, and search in the controlled partition

At the top of Listing 1 are three declarations for working memory element classes. Structures are similar to literals in OPS5. Unlike OPS5, though, PRIOPS is strongly typed. Fields of working memory elements may be of type *int, float, symbol,* or a *set of symbols.* Strong typing avoids run-time type testing and conversions, which is particularly important for automatic productions. A symbol is similar to a LISP symbol, and is loosely comparable to a string type in other languages. Each distinct symbol is stored uniquely and referenced via a unique pointer. Symbol comparisons, like integer and floating point comparisons, occur using basic machine operations. Consequently symbol tests are O(1) operations and do not depend upon symbol lengths. The CP alone acquires new symbols, but both partitions can use them.

<INSERT LISTING 1 ABOUT HERE.>

The current *sensors* element registers objects in the maze (WALL, HUMAN or EXIT) and their location relative to the PROGRAM. Each *visited* element records how often the PROGRAM has visited its location. Maze search rules and learning rules examine visited memory. Finally, inspect elements direct the immediate search from the PROGRAM's current location.

The rest of Listing 1 shows two controlled productions that help determine the PROGRAM's next move. They use both immediate sensory information (giving the current location of the PROGRAM) and accumulated memories. They trigger additional productions (not shown), implementing a heuristic strategy. First the PROGRAM rules out such undesirable moves as bumping into WALLS, heading down known dead ends, or reversing its walk. Then it prefers less visited locations and more distant walls. Finally it makes an arbitrary pick from remaining possible moves. Priority values in the controlled range order these heuristics sequentially.

Production *newly-arrived* initializes a visited record. Its left-hand-side determines that there is no visited for the current location of the sensors, so its right-hand-side action creates a visited record with a *visits* count of zero.

Production *look-out* triggers when a new sensors element is made at some visited location. Each line of tests on the left-hand-side of *look-out* is called a *condition element.* PRIOPS compiles condition elements for use by the Rete pattern matcher. When the matcher finds a visited element for the current sensors location, and finds that there are *no* inspect records (the third condition element is *negated*, signifying a successful match only in the *absence* of any memory elements matching this condition element), the matcher adds an executable instance of this production to the *conflict set.* (Existing inspect records would be left over from search preceding the most recent move; look-out must not trigger until garbage collection productions remove these inspect directives, hence the negated condition element.) Explicit priority and other criteria such as recency and specificity of matched memory elements identify a triggered production to execute. (Readers should consult Cooper and Wogrin(2) for more information on the OPS-based production inference cycle and conflict resolution strategies.) The right-hand-side of *look-out* first makes a *sequence* element (declaration not shown) to serialize the upcoming search. The next two actions compute and store an incremented *visits* count. The *bind* operations compute locations for the four adjoining spaces, and the *make* operations assert *inspect* elements. These elements direct upcoming search productions to inspect the adjoining spaces. The sequence element advances from value *eliminate-impossible* of Figure 1 through values *eliminate-deadends, eliminate-loops, eliminate-visited, eliminate-old, prefer-long,* and *ready-move,* firing productions that eliminate inspect proposals as it goes. Sequencing productions modify the sequence element as it completes its work at each stage. When only equally preferable inspect directives remain, an arbitrary one triggers controlled movement. After movement, controlled garbage collection productions remove the outdated sequence and remaining inspect records. The cycle beginning at newly-arrived and look-out of Figure 1 then runs again.

The majority of production processing time is spent matching memory elements to left-hand-side tests. Condition element tests come in two varieties. The first, *intra-condition element tests,* perform simple comparisons of memory element fields to constant values, and simple comparisons of element fields to other fields within the same working memory element. O(1) tests include *equality, inequality,* and basic *numerical order comparisons.* Controlled productions can use additional, O(n) tests such as *variable-length string order comparisons.* With all of the former tests being constant-time bounded, and with a compile-time constant specification of the number of tested fields (the program source), all automatic intra-condition element test sequences are constant-time bounded. Please consult our earlier papers for illustrations of the details of this matching process.(16,17)

However, the tests of *newly-arrived* and *look-out* contain no intra-condition element tests. Instead, they are *inter-condition element tests,* representing *joins* of two or more working memory elements. Rete performs the intra-condition element tests for a working memory element before attempting to join it to other elements matched by other condition elements. In the case of *look-out,* no intra-condition constraints appear for sensors, visited, or inspect elements. The result is that the matcher will perform the join tests, comparing the $\hat{}x$ and $\hat{}y$ locations fields, for each possible sensors-visited pair. In this program there is normally only one sensors element, but as a sensors element is made or updated, the join tests of *look-out* (and all controlled productions that join sensors and visited elements) are applied between the sensors element and *all* visited elements in working memory. After some maze exploration there will be many visited elements subject to these tests. The absence of any compile-time limit to the number of memory elements tested by a join through the abutment of two condition elements, is the fundamental source of time indeterminacy in Rete matching.(8)

4.3 Reactive movement in the automatic partition

Automatic partition matching trades expressive power for run-time efficiency and reduced computational complexity.(14) By compiler fiat, at most one working memory element can match an automatic condition element. Each successfully matched working memory element discards any prior match. In Rete terms, each condition element in an automatic production contributes at most one memory element to a join. Such a join reduces to a constant-length sequence of simple O(1) tests. Prohibiting memory growth eliminates the time indeterminacy of variable-sized joins.

Listing 2 shows three automatic productions from the maze program. The first, *panic-left-up,* detects a human to the left. It also determines there is an avenue up from its current location that is further from a wall than the competing down avenue. Once fired, *panic-left-up,* removes the sensors element so that no other production can react to it. It then makes a *move* element for upward movement, which in turn triggers the second production shown, *automove.*

<INSERT LISTING 2 ABOUT HERE.>

With only one condition element, *panic-left-right* matching consumes no join time. All tests are intra-element tests. Note that the tests of this production are extremely specialized. Most automatic productions have specialized intra-element tests because these tests must safeguard desired matches from being overwritten. Remember that a new match eliminates any previous match to a condition element. By the time matching gets to inter-element join tests, the contributing memory elements are unique. The intra-element tests select each condition element's unique memory element for join matching.

Leading intra-element test sequences common to more than one condition element execute only once for a single working memory element. Only when intra-condition element test sequences diverge are distinct match tests compiled. The result is that automatic intra-condition element tests are compiled into a decision tree, with a single place register at each leaf storing a reference to the matching working memory element. For instance, *panic-left-up* and the third production of Listing 2, *panic-left-right,* share the initial test *sensors ˆleft-sense human.* When the HUMAN appears to the PROGRAM's left sensors, this test executes only once, with the two productions' testing diverging after that point. Both productions may trigger, but when the first to execute removes the sensors element, the alternative production will no longer trigger.

Production *automove* joins two working memory elements. There are no inter-condition element restrictions, so the most recent *move* request with an urgency > 0 combines with the most recent *lastmove* record to trigger the automatic move. *Automove* removes the *move* element, updates *lastmove,* and calls the C language move driver. A controlled production with this left-hand-side could match multiple move request / lastmove record joins. Since *automove* is an automatic production, it can only match the most recent elements satisfying its tests. It responds to the immediate sensory and control information in constant time.

4.4 Garbage collection at the controlled-automatic interface

Controlled productions that search and learn can generate redundant and outdated working memory elements that constitute garbage. Also, automatic productions can generate sensory and movement messages that are buffered in the CP. Outdated messages in these buffers constitute garbage as well. The classical approach to garbage collection in LISP systems is to bring the system to a halt when storage capacity is exhausted, and recover all reclaimable memory during this halt interval.(6) Clearly such an approach is not suited to a reactive, time-constrained system. *Incremental garbage collectors* (sometimes called *real-time* collectors) avoid halting for storage reclamation by distributing reclamation activities in small, constant-time bound pieces across all calls for storage allocation.(6,15,23) The problem with such systems is that they exact a small penalty from all processes, both those with and those without O(1) execution requirements, for storage reclamation activities.

PRIOPS performs garbage collection by using mechanisms built into controlled matching and by using explicit controlled reclamation productions at the controlled-automatic interface - priority 0 productions. The PRIOPS reclamation strategy avoids penalizing automatic processing. A ready list of initialized working memory elements is available for automatic productions that send messages into controlled buffers. During automatic execution productions remove initialized elements from the ready list and add message elements to controlled buffers in O(1) time. When automatic execution completes, the controlled matcher replenishes this ready list to prepare for the next burst of automatic activity The explicit garbage collection productions recover buffered memory elements that, through testing of contents and time stamps, are found to be redundant or outdated. Replenishing the ready list places an average real-time requirement on controlled storage management, since the ready list must contain sufficient elements to satisfy the most demanding burst of automatic activity. The size of the ready list is coded directly into the PRIOPS program. Dynamic adjustment of ready list size is an area for future work.

Listing 3 shows an explicit garbage collection production. *Collect-old-move* recovers an older *move* element after a newer one is posted. Remember that an automatic production such as *automove* from

Listing 2 will trigger only on the most recently matched move element, so this garbage collection is necessary only for the CP. Standard OPS5 and controlled PRIOPS conflict resolution strategy results in an execution for *collect-old-move* that binds the most recent move element to the first condition element, and the older move element to the <*older*> second condition element, which *collect-old-move* removes from working memory. This collection occurs after all automatic use of working memory elements, including the most recent move, is done.

<INSERT LISTING 3 ABOUT HERE.>

4.5 Learning and learned productions

Learning rules build automatic productions for escape in subsequent traversals of the current maze. Learning rules execute only upon PROGRAM exit from the maze. They examine visited records in working memory and write production definitions to a file for later compilation. Their priority allows garbage collection productions to work before commencing learning.

The maze program learns an escape route by using a primitive form of production *chunking.* Chunking uses knowledge acquired during a problem solving episode to build specialized productions that embody that knowledge, thereby avoiding comparable searches in the future. In the *SOAR* production system, chunking is an inherent component of the architecture.(12) SOAR uses both domain knowledge and generic weak search methods to search problem spaces, and automatically builds chunk productions that avoid repeated searches.(11,13) The PRIOPS maze program uses maze-specific productions to build its chunks.

Learned productions trigger on the present location of the PROGRAM. Only when the HUMAN blocks the escape route will these productions fail to trigger in a learned maze. At other times they preempt all other matching and actions in the production system. These productions are of course not present for a novel maze.

Listing 4 shows learning production *learn-pass-up*, followed by the learned escape production *auto-24-1-up*.

<INSERT LISTING 4 ABOUT HERE.>

Maze learning proceeds after a successful escape by performing a depth-first search from the EXIT along paths traversed by the PROGRAM. A *learn* memory element directs the search similarly to the *inspect* elements for initial maze exploration from Listing 1. The left-hand-side of *learn-pass-up* joins a *learn* and *visited* record to build a link from the visited element's location to a location already examined in the learning search. For this production, the original escape route included an *up* move from the visited location to an adjoining location. Join tests match the *ˆx* and *ˆy* fields of the joined memory elements. The *ˆvisits < 10000* test prunes locations that are part of dead-end paths. Dead-end detection code marks visited records with a ˆvisits value of 10000 during the original maze search.

The *write* actions on the right-hand-side of *learn-pass-up* generate production text such as that of *auto-24-1-up* in Listing 4. *Learn-pass-up* builds the location into this learned automatic production, making it extremely specific. At priority 127, its matching preempts all other matching and, when matching succeeds, any other queued processing remains queued while the right-hand-side of *auto-24-1-up* executes. When *auto-24-1-up* moves the PROGRAM to an adjacent location that triggers another priority 127 learned production, all lower priority processing continues to wait. *Auto-24-1-up* shares its initial *ˆup-sense* test with all other *auto-X-Y-up* learned productions. In addition, this production shares the *ˆup-sense <> human ˆx 24* test sequence with all other *auto-24-Y-up* productions. Recent work on chunking in SOAR has concentrated on trading generality for matching speed by generating efficient *unique-attribute chunks* instead of generalized *multi-attribute chunks* that create large join cross-products.(22) PRIOPS automatic productions take this approach to an extreme, trading generality for both speed and *speed predictability* by compiling O(1) *unique-event* chunks. The next section demonstrates performance gains achieved by learning automatic reactions.

4.6 Maze statistics

We ran the PRIOPS maze program on an AT&T PC-6300, which contains an 8 MHz. 8086 processor and an 8087 math coprocessor. PRIOPS was compiled using Microsoft® C 5.1 with the "/Oti /Gs /AL /FPi" compile switches for optimization, memory management, and floating-point library selection; the maze application itself uses no floating point processing. The current PRIOPS compiler generates intermediate code for both Rete matching and right-hand-side execution. A planned modification to PRIOPS will support generation of C source code for compilation to native run-time code. PRIOPS comprises about 12,000 lines of C code.

An additional 800 lines of C code support the maze environment, and the maze's PROGRAM itself is coded in 940 lines of PRIOPS productions. Referring to Figure 3, there are 2 initialization, 9 garbage collection, 2 declarative memory maintenance, 31 controlled maze search, 1 controlled move, and 14 learning productions. There are 24 HUMAN avoidance, 5 EXIT approach, and 1 automatic move productions. Learning acquires 116 automatic productions of the type in Listing 4 for the maze of Figure 2.

With no interference from the HUMAN, the PROGRAM takes 909 seconds and 566 moves (1.6 seconds/move) to discover a 104 step path to the EXIT for the maze of Figure 2. Post-exit learning uses an additional 172 seconds.

Learning automatic behavior improves performance dramatically. Automatic escape productions reach the EXIT in 34 seconds and 104 moves (.33 seconds/move). Then the PROGRAM sits at the EXIT for 21 seconds running matching for deferred production tests. Listings 1 and 4 help illustrate the source of this delay. Note that the two controlled productions of Listing 1 (and many other controlled productions as well) trigger on *sensors* elements. Now examine *auto-24-1-up* in Listing 4; it, too, matches *sensors* elements. During learned-production escape from the maze, priority 127 matching of productions such as *auto-24-1-up* defers any controlled matching required for *sensors* elements. Only when the PROGRAM reaches the EXIT, and no automatic productions are firing preemptively, can deferred controlled matching execute. This deferred matching for productions such as *newly-arrived* causes the 21 second delay. Optimal coding of controlled productions would eliminate this delay, but the point here is to illustrate the effect of priority-based matching deferral in PRIOPS.

If we change the priority of learned productions like *auto-24-1-up* to 0, thereby placing them into the CP, the escape statistics change. Now learned escape, acting in the CP, takes 64 seconds and 104 moves (.62 seconds per move), and 2 seconds for termination. There is no deferred matching, since all controlled matching completes before CP conflict resolution. It is also noteworthy that the total time for automatic escape is 11 seconds less than the total time for controlled escape. How does automatic matching achieve this speedup? By combining assertions and retractions of working memory elements where possible. Since automatic matching allows only one memory element-to-condition element match per distinct condition element, only the most recent memory element assertion or retraction is queued at each automatic matching step. Suppose memory element *A* has matched condition element *X.* Then a more recently asserted memory element *B* also matches *X.* The match scheduling algorithm will abort the match of *A* in joins of *X.* Condition element *X* can only match the single most recently asserted memory element that satisfies its tests, in this example element *B.* Only *B* participates in subsequent joins. The AP insistence on bounded resources forces it to reuse memory, thus avoiding some search as well as garbage collection. Controlled matching cannot use this enhancement, since multiple memory elements may match a single condition element at a given time. Therefore the PRIOPS AP provides not only O(1) matching with preemptive priorities, but also provides opportunities for additional speed enhancements beyond the capabilities of standard Rete.

5. CONCLUSIONS

Although we have tested PRIOPS using only simple experimental problems to date, we feel confident that it will scale to meet the needs of genuine knowledge-based embedded applications. The synthesis of O(1) preemptive match processing and the production system notation provides the ability to specify important environmental conditions and timely reactions at a more appropriate level of abstraction for knowledge-based problems than is provided by traditional embedded software techniques. The fact that humans use a comparable two-tiered approach to timely interaction with physical environments indicates that the PRIOPS fundamental approach is correct. Research in learning, both in human automatic process acquisition and production system learning mechanisms such as chunking, offers promise of methods for machine construction of appropriate automatic productions.

Future work on PRIOPS will include research on dynamic learning. The maze learning discussed here relies on hand coding of application-specific, ad hoc learning productions. Human automatic behavior acquisition relies almost entirely on practice. Learning should involve incremental refinement of controlled processing into automatic productions. Other planned work includes construction of tools for analysis of automatic partition data flow and more complete examination of activities belonging to the controlled-automatic interface.

REFERENCES

1. Gul Agha, "Global Synchrony and Asynchrony," Section 2.2, *Actors - A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA: MIT Press, 1986, p. 9-11.

2. Thomas A. Cooper and Nancy Wogrin, *Rule-based Programming with OPS5,* San Mateo, Ca: Morgan Kaufmann, 1988.

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

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

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

6. Richard P. Gabriel, "Memory Management in LISP," *AI Expert*, Vol. 2, No. 2 (February, 1987), p. 32-38.

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

8. 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.

9. Ivar Jacobson, "Language Support for Changeable Large Real Time Systems," *OOPSLA '86 Conference Proceedings,* ed. Norman Meyrowitz. New York, NY: ACM, 1986, p. 377-384.

10. Thomas J. Laffey, Preston A. Cox, James L. Schmidt, Simon M. Kao and Jackson Y. Read, "Real-Time Knowledge-Based Systems," *AI Magazine,* Vol. 9, No. 1 (Spring, 1988), p. 27-45.

11. John E. Laird and Allen Newell, "A Universal Weak Method." Memo CMU-CS-83-141, Carnegie-Mellon University, June, 1983.

12. John E. Laird, 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.

13. John E. Laird, Allen Newell and Paul S. Rosenbloom. "SOAR: An Architecture for General Intelligence." Report No. STAN-CS-86-1140, Stanford University, December, 1986.

14. Hector J. Levesque and Ronald J. Brachman, "A Fundamental Tradeoff in Knowledge Representation and Reasoning (Revised Version)," *Readings in Knowledge Representation,* ed. Ronald J. Brachman and Hector J. Levesque, Los Alstos, Ca: Morgan Kaufmann, 1985, p. 42-70.

15. Henry Lieberman and Carl Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of the ACM,* Vol. 26, No. 6 (June, 1983), p. 419-429.

16. Dale E. Parson 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.

17. Dale E. Parson and Glenn D. Blank, "Automatic versus controlled processing: an architecture for

real-time production systems." To appear in *International Journal of Expert Systems: Research and Applications,* Vol. 2, No. 4 (early 1990).

18. 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.

19. 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.

20. 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.

21. Milind Tambe, 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.

22. Milind Tambe and Paul Rosenbloom, "Eliminating Expensive Chunks by Restricting Expressiveness," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* Vol. 1, August, 1989, p. 731-737.

23. David Ungar and Frank Jackson, "Tenuring Policies for Generation-Based Storage Reclamation," *OOPSLA '88 Conference Proceedings,* ed. Norman Meyrowitz. New York, NY: ACM, 1988, p. 1-17.

24. Niklaus Wirth, "Toward a Discipline of Real-Time Programming," *Communications of the ACM,* Vol. 20, No. 8 (August, 1977), p. 577-583.

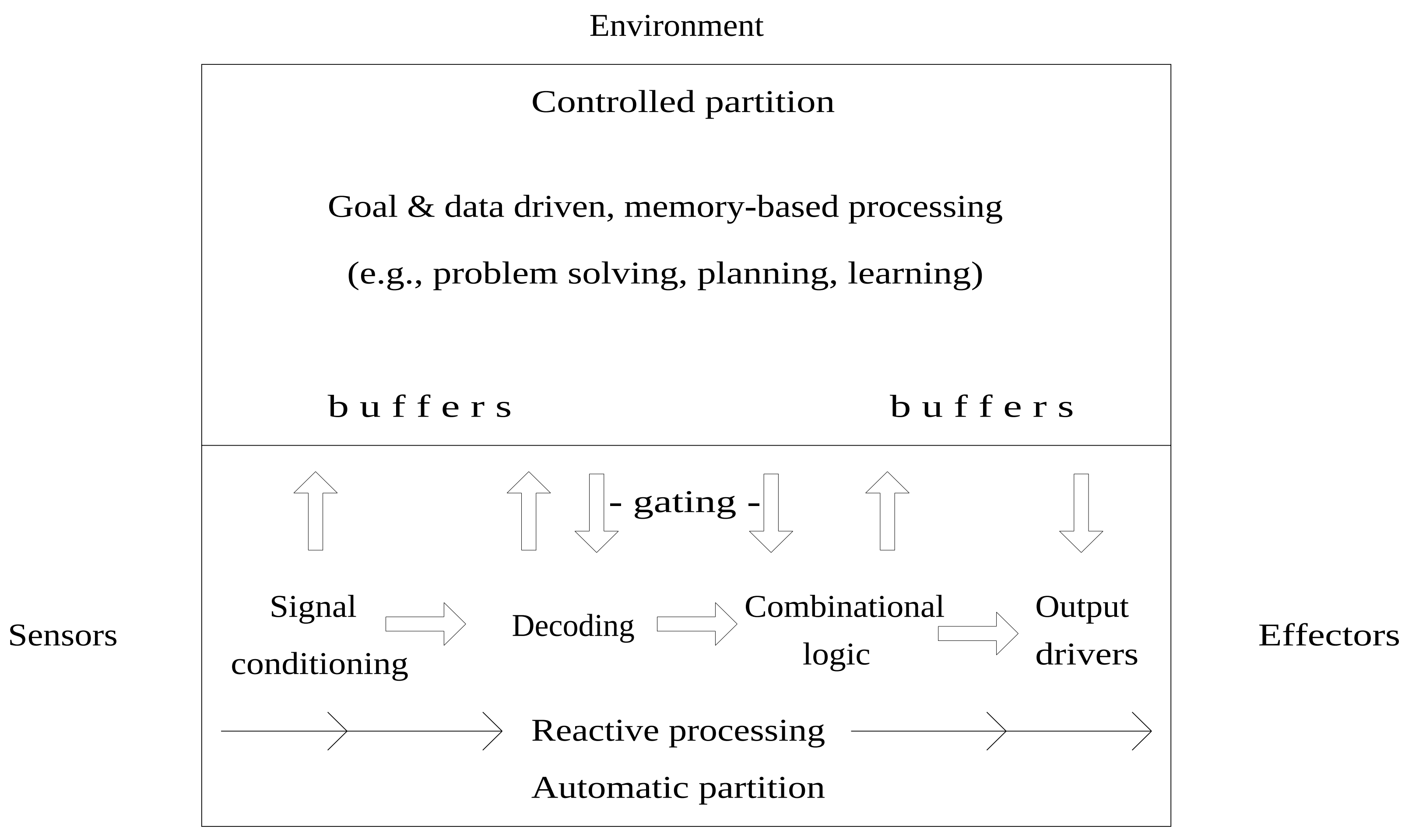

Figure 1 - The two-tiered, controlled-automatic architecture

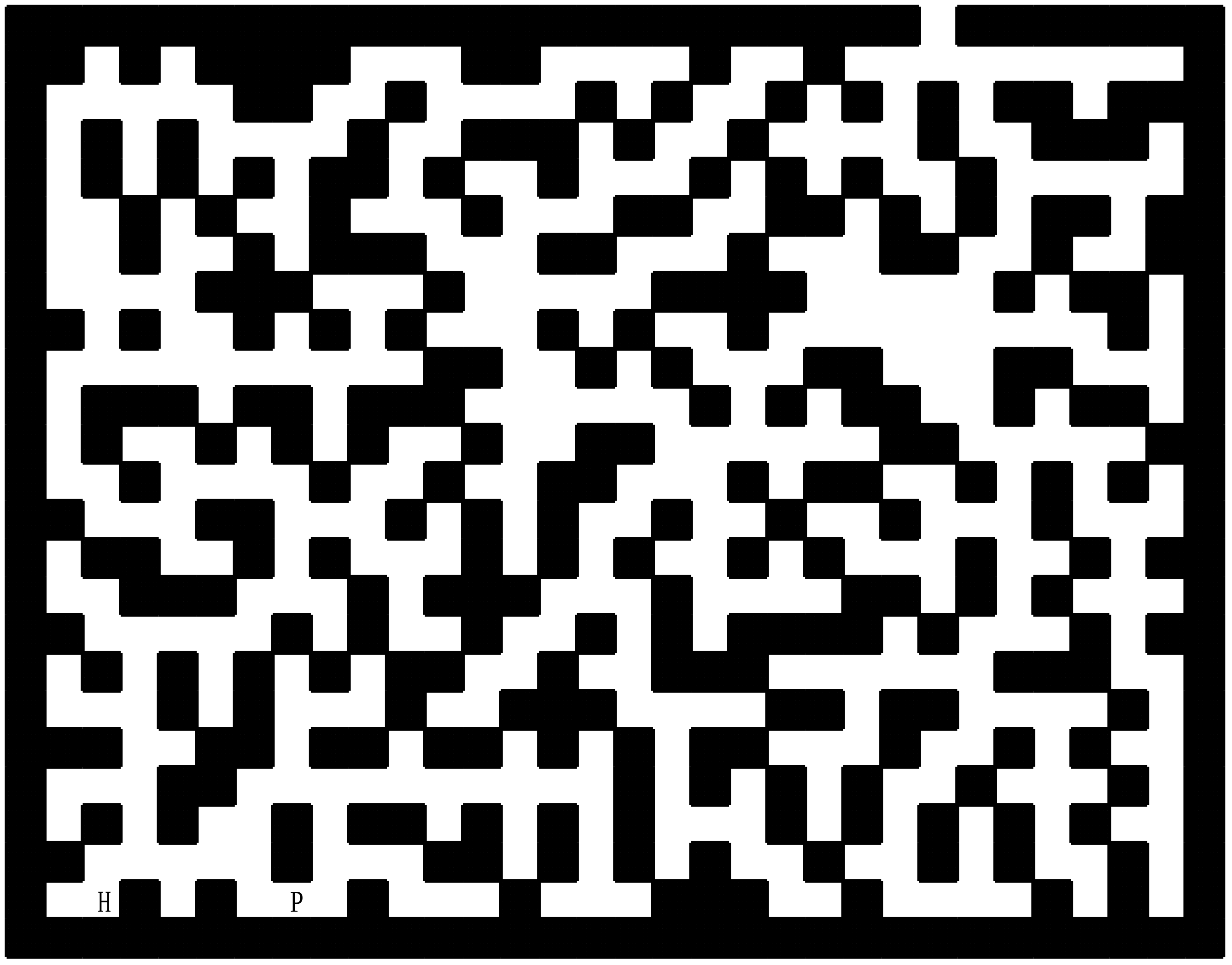

Figure 2 - The maze

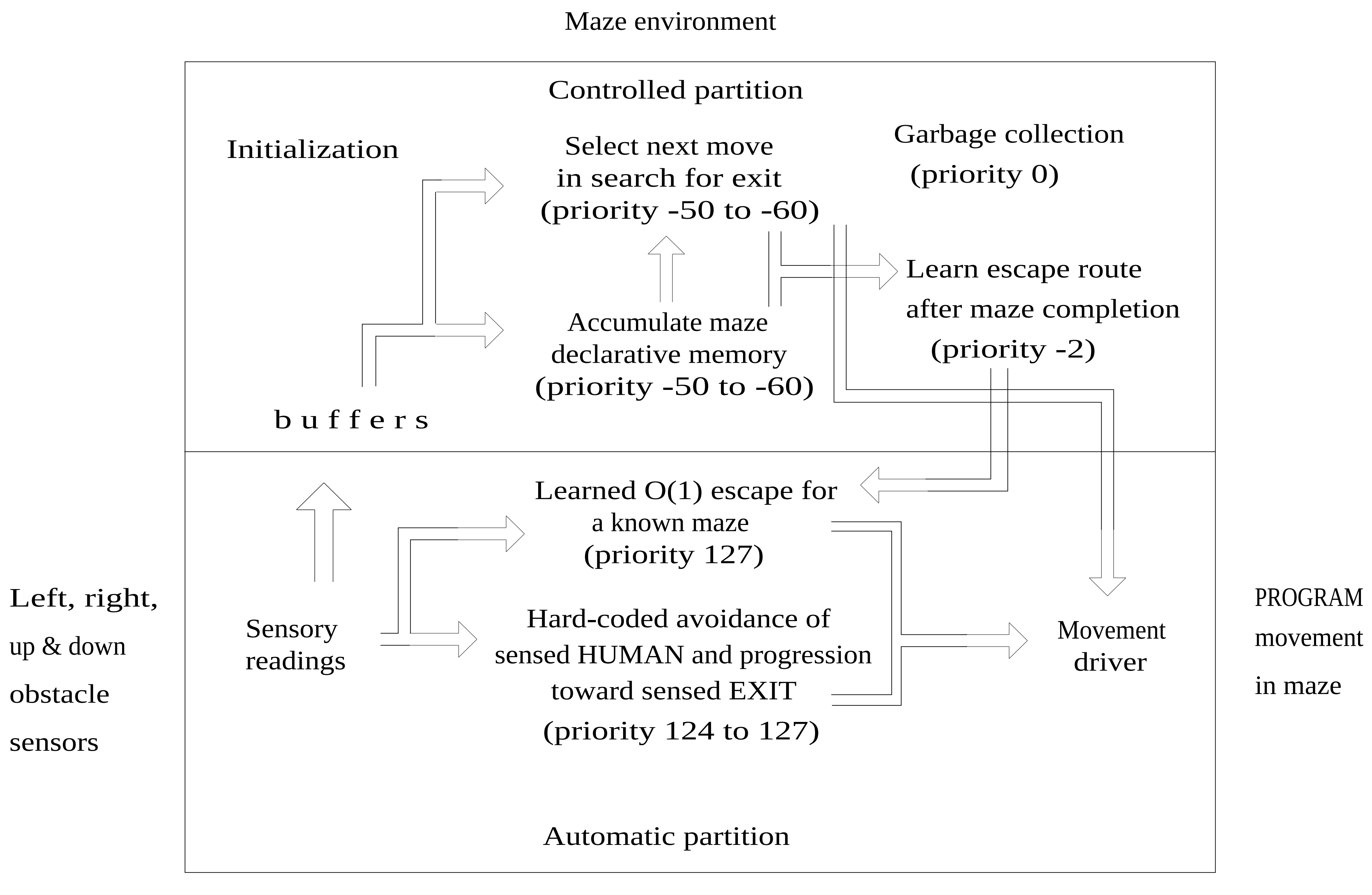
Maze environment
Controlled partition
Initialization
Select next move in search for exit (priority -50 to -60)
Garbage collection (priority 0)
Learn escape route after maze completion (priority -2)
Accumulate maze declarative memory (priority -50 to -60)
b u f f e r s
Learned O(1) escape for a known maze (priority 127)
Left, right, up & down obstacle sensors
Sensory readings
Hard-coded avoidance of sensed HUMAN and progression toward sensed EXIT (priority 124 to 127)
Movement driver
PROGRAM movement in maze
Automatic partition

Figure 3 - PROGRAM data flow during maze traversal

```
(structure sensors              ; 4 pseudo-sendors of program
         symbol  left-sense     ; object-to-left: WALL or HUMAN or EXIT
         int      left-distance
         symbol  right-sense   int   right-distance
         symbol  up-sense      int   up-distance
         symbol  down-sense    int   down-distance
         int      x            int   y ; current PROGRAM x,y location
)

(structure visited              ; remembered history of a location
         int x int y            ; location
         int visits             ; number of visits to here
)

(structure inspect              ; possible single move destination
         int x int y            ; location of proposed move
         symbol direction       ; direction from current sensor
)

(p newly-arrived -60
         (sensors ^x <myx> ^y <myy>)
         - (visited ^x <myx> ^y <myy>)
         ->
         (make visited ^x <myx> ^y <myy> ^visits 0)
)

(p look-out -60
         (sensors ^x <myx> ^y <myy>)
         {(visited ^x <myx> ^y <myy> ^visits <seen>) <books>}
         - (inspect)
         ->
         (make sequence ^current eliminate-impossible) ; 1st step of search
         (bind <newseen> (compute <seen> + 1))
         (modify <books> ^visits <newseen>) ; update memory of loc. myx,myy
         (bind <left> (compute <myx> - 1))
         (bind <right> (compute <myx> + 1))
         (bind <up> (compute <myy> - 1))
         (bind <down> (compute <myy> + 1))
         (make inspect ^x <left> ^y <myy> ^direction left) ; init. the search
         (make inspect ^x <myx> ^y <up> ^direction up)
         (make inspect ^x <right> ^y <myy> ^direction right)
         (make inspect ^x <myx> ^y <down> ^direction down)
)
```

Listing 1 - Declarations and controlled productions for a new maze location

--

--

```
(p panic-left-up 125 ; prefer right angle turns from human
        {(sensors ^left-sense human
                ^up-distance {<escape> > 1} ^down-distance <= <escape>)
                <sense>}
        ->
        (remove <sense>)
        (make move ^direction up ^urgency 125)
)

(p automove 127
        {(move ^urgency > 0 ^direction {<way> <> nil}) <moving>}
        {(lastmove) <former>}
        ->
        (modify <moving> ^direction nil ^urgency 0)
        (modify <former> ^direction <way>)
        ; each "call move" generates a new "sensors" element
        (call move <way>)
)

(p panic-left-right 124
        {(sensors ^left-sense human ^right-distance > 1) <sense>}
        ->
        (remove <sense>)
        (make move ^direction right ^urgency 125)
)
```

Listing 2 - Two automatic productions for emergency reactions

```
(p collect-old-move 0                    ; 2 moves shown, 1 outdated
          (move ˆdirection <way>)
          {(move ˆdirection <> <way>) <older>}
          ->
          (remove <older>)
)
```

Listing 3 - A priority 0, controlled garbage collection production

```
(p learn-pass-up -2 ; translate declarative memory into auto production
        {(learn ^status learning ^direction up
                ^fromx <endx> ^fromy <endy>) <passed>}
        {(visited ^visits < 10000 ^x <endx> ^y <endy>) <memory>}
        ; spot was visited
        ->
        ; first write the learned production
        (write learnfile "(p auto-" <endx> "-" <endy> "-up 127\n")
        (write learnfile "{(sensors ^up-sense <> human ^x "
                <endx> " ^y " <endy> ") <sensing>}\n")
        (write learnfile "{(lastmove) <oldmove>}\n")
        (write learnfile "->\n")
        (write learnfile "(remove <sensing>)\n")
        (write learnfile "(modify <oldmove> ^direction up)\n")
        (write learnfile "(call move up)\n)\n\n")
        ; learned production written, now make this the new dest.
        (remove <memory>) ; not needed any longer
        (modify <passed> ^status done)
        (bind <up> (compute <endy> - 1))
        (bind <down> (compute <endy> + 1))
        (bind <left> (compute <endx> - 1))
        (bind <right> (compute <endx> + 1))
        (make learn ^status learning ^fromx <endx> ^fromy <up>
                ^direction down)
        (make learn ^status learning ^fromx <endx> ^fromy <down>
                ^direction up)
        (make learn ^status learning ^fromx <left> ^fromy <endy>
                ^direction right)
        (make learn ^status learning ^fromx <right> ^fromy <endy>
                ^direction left)
)

(p auto-24-1-up 127
        {(sensors ^up-sense <> human ^x 24 ^y 1) <sensing>}
        {(lastmove) <oldmove>}
        ->
        (remove <sensing>)
        (modify <oldmove> ^direction up)
        (call move up)
)
```

Listing 4 - Maze escape learning and a learned escape automatic production